Contrôle statistique du processus de propagation de contraintes

Frédéric Boussemart **Fred Hemery** **Christophe Lecoutre** **Mouny Samy Modeliar**
CRIL - CNRS UMR 8188, Université Lille Nord de France, Artois, France
{boussemart,hemery,lecoutre,modeliar}@cril.fr

Résumé

Dans cet article, nous nous intéressons à la propagation de contraintes, un processus de filtrage qui est typiquement exécuté à chaque étape d'un algorithme de recherche en profondeur d'abord avec retours-arrière tel que MAC. A partir d'une analyse statistique portant sur des propriétés précises du mécanisme de propagation, nous montrons qu'il est possible d'effectuer des prédictions fiables sur la capacité du processus de propagation à détecter l'incohérence. En utilisant cette observation dans le but de contrôler l'effort de propagation, nous montrons son intérêt pratique sur un jeu d'essai comportant de nombreuses séries d'instances CSP.

Abstract

In this paper, we investigate constraint propagation, a mechanism that is run at each basic step of a backtrack search algorithm such as the popular MAC. From a statistical analysis of some relevant features concerning propagation on a large set of CSP instances, we show that it is possible to make reasonable predictions about the capability of constraint propagation to detect inconsistency. Using this observation in order to control propagation effort, we show its practical effectiveness.

1 Introduction

De nombreux problèmes de décision sont de nature combinatoire, et peuvent être modélisés à l'aide de variables discrètes combinées par des contraintes. On obtient alors des instances du problème de satisfaction de contraintes (CSP pour Constraint Satisfaction Problem) qui sont représentés formellement par des réseaux de contraintes (CN pour Constraint Network) [9, 12]. Résoudre une instance CSP (ou résoudre un CN) consiste (dans cet article) à trouver une solution ou prouver qu'aucune solution n'existe.

La recherche avec retours-arrière (backtrack search) est utilisée couramment pour résoudre les instances CSP. Pour ce type de recherche, une exploration en profondeur d'abord permet d'instantier les variables et un mécanisme de retours-arrière permet de gérer les impasses détectées. Après chaque instantiation de variable, un processus de filtrage est exécuté sur la base d'une propriété locale, appelée cohérence (consistency), qui permet d'élaguer certaines parties de l'espace de recherche ne contenant aucune solution. La cohérence d'arc généralisée (GAC pour Generalized Arc Consistency) est la forme de cohérence locale la plus forte lorsque les contraintes sont considérées de manière indépendante. Intuitivement, GAC permet d'éliminer en toute sécurité une valeur a du domaine d'une variable x si il existe une contrainte c impliquant x mais n'acceptant aucun tuple (construit sur la base des domaines courants des variables) avec la valeur a pour x. Aussitôt qu'une inférence locale (par exemple, correspondant à l'élimination d'une valeur) est effectuée, les conditions pour déclencher de nouvelles inférences peuvent potentiellement être vérifiées puisque les variables sont typiquement partagées par plusieurs contraintes. Cette manière de propager le résultat d'inférences locales de contraintes en contraintes est appelée *propagation de contraintes* (par exemple, voir [2]).

L'un des algorithmes de recherche avec retours-arrière les plus utilisés est appelé MAC (Maintaining Arc Consistency) [21]. Pendant la recherche, MAC maintient en permanence GAC de manière à réduire la taille de l'arbre de recherche à explorer. GAC permet d'éliminer de nombreuses valeurs inutiles, mais malheureusement établir cette propriété constamment en cours de recherche peut s'avérer très coûteux. FC [11] est un autre algorithme de recherche avec retours-arrière qui réalise une forme très limitée de propagation de contraintes : seules les contraintes impliquant la dernière variable assignée sont "propagées". Bien que MAC soit considérée comme état de l'art, sur certains problèmes, FC peut parfois surclasser MAC.

Plusieurs tentatives pour ajuster le bon niveau de filtrage ont été effectuées ces dernières années. Tout d'abord, dans [8], les auteurs proposent une vison paramétrée de la cohérence locale qui permet de nombreuses formes partielles de GAC d'être établies. Le paramètre utilisé pour cette approche est la distance entre la dernière variable assignée et les autres variables dans le graphe de contraintes. Ensuite, Mehta et van Dongen ont introduit une approche probabiliste [20] de manière à n'effectuer uniquement que les tests de contraintes utiles en ne recherchant pas de support pour une valeur lorsqu'il y a une forte probabilité qu'un tel support existe. Cette technique d'inférence de support probabiliste a été étudiée à la fois pour GAC et SAC (singleton arc consistency). Plus récemment, Stergiou [23] a proposé d'adapter dynamiquement le niveau de cohérence locale devant être appliquée durant la recherche. Il a montré que l'information concernant les domaines devenus vides (domain wipe-outs) et les suppressions de valeurs pendant la recherche peuvent être utilisées non seulement pour sélectionner les variables mais également pour ajuster automatiquement le niveau de filtrage

Dans cet article, à partir d'une analyse de nature statistique concernant la propagation de contraintes, nous montrons qu'il est possible de faire des prédictions raisonnables sur la capacité de la propagation à détecter l'incohérence. Par une corrélation existant entre deux critères importants de la propagation (établie à partir d'un large jeu d'essai), à savoir la longueur et le résultat de la propagation, nous montrons qu'il est possible de limiter le coût de la propagation de contraintes de manière originale. Cette propriété se révèle prometteuse, illustrée ici avec une variante de MAC.

2 Préliminaires

Un réseau de contraintes discret (CN) P est composé d'un ensemble fini de variables, noté $vars(P)$, et d'un ensemble fini de contraintes, noté $cons(P)$. Chaque variable x possède un domaine, noté $dom(x)$, qui contient l'ensemble fini des valeurs qui peuvent être assignées à x. Chaque contrainte c porte sur un ensemble ordonné de variables, appelé portée et noté $scp(c)$. L'ensemble des tuples autorisés pour les variables impliquées dans c est défini formellement par une relation notée $rel(c)$, qui peut être donnée en extension ou en intention. L'arité d'une contrainte c est la taille de $scp(c)$. Une contrainte binaire porte sur exactement 2 variables, et une contrainte non-binaire sur plus de 2 variables.

Les CN sont utiles pour représenter de nombreux problèmes combinatoires comme, par exemple, le problème de coloration de graphe : étant donné un graphe $G = (V, E)$ et k couleurs, est-il possible d'associer une couleur à chaque sommet tel que les deux sommets de chaque arête soient coloriés différemment ? Ceci peut être modélisé par un CN P tel que (i) pour chaque sommet $v \in V$, il existe une variable x_v dans $vars(P)$ avec $dom(x) = \{1, 2, \ldots, k\}$, et (ii) pour chaque arête $e = \{v, v'\} \in E$, il existe une contrainte c_e dans $cons(P)$ tel que $scp(c_e) = \{x_v, x_{v'}\}$ et sa sémantique est donnée par $x_v \neq x_{v'}$.

Une solution à un CN est l'assignation d'une valeur à chaque variable telle que toutes les contraintes soient satisfaites. Un CN est dit *cohérent* ssi il admet au moins une solution. Le problème de satisfaction de contraintes (CSP) consiste à déterminer si un CN donné est cohérent ou non. Aussi, une instance CSP est définie par un CN qui est résolu en trouvant une solution ou en prouvant qu'aucune solution n'existe. Dans de nombreux cas, une instance CSP peut être résolue par des techniques de recherche et d'inférence [9, 12].

Une cohérence locale est une propriété (ou condition) générale définie sur les CN. Typiquement, la non vérification d'une cohérence locale nous permet de faire des déductions, également appelées inférences, qui révèlent des nogoods (standards), i.e. révèlent des instantiations (partielles) de variables qui ne peuvent mener à des solutions. La cohérence d'arc généralisée (GAC) est la propriété centrale en programmation par contraintes.

Étant donné un ensemble ordonné $\{x_1, \ldots, x_i, \ldots, x_r\}$ de r variables et un r-tuple $\tau = (a_1, \ldots, a_i, \ldots, a_r)$ de valeurs, la valeur a_i sera notée $\tau[x_i]$. Un r-tuple τ est *valide* sur une contrainte c d'arité r ssi $\forall x \in scp(c)$, $\tau[x] \in dom(x)$. Rappelons qu'un tuple τ est *autorisé*, ou accepté, par une contrainte c ssi $\tau \in rel(c)$. Un r-tuple τ est un *support* sur une contrainte c d'arité r ssi τ est un tuple valide sur c qui est également autorisé par c. Si τ est un support sur une contrainte c impliquant une variable x et tel que $\tau[x] = a$, on dit alors que τ est un support pour (x, a) sur c ; on dit également que (x, a) est supporté par c.

Une v-valeur d'un CN P est un couple variable-valeur (x, a) tel que $x \in vars(P)$ et $a \in dom(x)$. Une v-valeur (x, a) d'un CN P est *GAC-cohérente* sur P ssi pour toute contrainte c de P impliquant x, il existe un support pour (x, a) sur c. Une contrainte c est *GAC-cohérente* ssi $\forall x \in scp(c), \forall a \in dom(x)$, il existe un support pour (x, a) sur c. Un CN P est *GAC-cohérent* ssi chaque contrainte de P est GAC-cohérente. Si une v-valeur (x, a) n'est pas GAC-cohérente, elle est dite *GAC-incohérente*. Il est aisé de constater que toute v-valeur GAC-incohérente ne peut apparaître dans aucune solution et, par conséquent, est (globalement) incohérente. Rétablir GAC signifie rendre le CN GAC-cohérent en éliminant toutes les v-valeurs GAC-incohérentes. Le résultat de l'établissement de GAC est un CN unique, noté $GAC(P)$, qui est appelé la *GAC-fermeture* de P ; voir par exemple, [2]. De nombreux algorithmes ont été proposés pour établir (G)AC. Comme exemples d'algorithmes génériques, citons (G)AC3 [17, 18], (G)AC2001 [4] et GAC3^{rm} [14]. Pour les contraintes exprimées en extension, on trouve des développements récents [16, 15, 10, 7, 13].

3 Propagation orientée variable

Pour établir une cohérence donnée sur un réseau de contraintes, des déductions locales (inférences) sont effectuées de manière itérative jusqu'à l'obtention d'un point fixe. Étant donné que les variables sont typiquement impliquées dans plusieurs contraintes, toute inférence effectuée peut en induire de nouvelles. Le mécanisme qui consiste à propager le résultat d'inférences locales de contraintes en contraintes est appelée propagation de contraintes, et est mis en oeuvre par des algorithmes de filtrage.

Classiquement, seuls les événements concernant les variables sont utilisés pour la propagation de contraintes. Dans un contexte de filtrage générique (à gros grain), qui exécute une même procédure indépendamment de la nature des contraintes, la modification du domaine d'une variable (suppression d'une ou plusieurs valeurs) est le seul type d'événement retenu. Le filtrage à grain fin est quant à lui guidé par les valeurs qui sont supprimées.

Dans cette section, nous donnons une description d'un schéma de propagation générique à gros grain qui peut être utilisé pour appliquer GAC sur un réseau de contraintes donné. Les algorithmes de propagation à gros grain effectuent des révisions successives d'arc. Un arc est représenté par un couple (c, x) où c est une contrainte et x une variable contenue dans $scp(c)$. La révision d'un arc (c, x) supprime de $dom(x)$ chaque valeur qui n'a pas de support sur c. Une révision est dite $effective$ si elle permet de supprimer au moins une valeur. Une révision est dite $inutile$ si on peut prédire qu'elle sera infructueuse. De manière évidente, il faut éviter d'effectuer des révisions inutiles.

Dans le *schéma orienté variable* [19, 24, 5], lorsqu'une valeur du domaine d'une variable est supprimée, la variable est ajoutée à un ensemble Q appelé *queue de propagation*. Le *schéma orienté variable* peut être optimisé par l'utilisation de tampons temporels (timestamps). Ceux-ci permettent de dater certains événements et de suivre au cours du temps la progression d'un algorithme. Ils permettent notamment dans notre cas de déterminer si une révision est inutile ou pas.

En introduisant une horloge (compteur) globale $time$ et en associant un tampon $stamp[x]$ à chaque variable x et un tampon $stamp[c]$ à chaque contrainte c, il est possible de déterminer les révisions qui seront effectives. La valeur $stamp[x]$ indique à quel moment a eu lieu la dernière suppression d'une valeur dans $dom(x)$ tandis que la valeur $stamp[c]$ indique le moment le plus récent où la contrainte c a été rendue GAC-cohérente. Les variables $time$, $stamp[x]$ pour chaque variable x et $stamp[c]$ pour chaque contrainte c sont initialisées à 0. La valeur de $time$ est incrémentée à chaque fois qu'une variable est ajoutée à Q et à chaque fois qu'une contrainte est rendue GAC-cohérente.

L'appel $\mathsf{GAC}^{var}(vars(P))$, voir l'algorithme 1, assure la cohérence d'arc généralisée pour un réseau de contraintes P donné ; dans un premier temps nous passons sous silence les instructions en gris clair des lignes 4,7 et 17-18. La valeur booléenne $false$ est retournée si P est prouvé GAC-incohérente, c'est-à-dire si $GAC(P) = \bot$. Soit $past(P)$ l'ensemble des variables passées de P, c'est-à-dire les variables de P qui ont été instanciées par un algorithme de recherche avec retours-arrière comme FC ou

Algorithme 1: $\mathsf{GAC}^{var}(X_{evt}$: ensemble de variables)

Résultat : $true$ iff $GAC(P) \neq \bot$

1 $Q \leftarrow \emptyset$
2 **foreach** *variable* $x \in X_{evt}$ **do**
3 insert(Q, x)
4 $cnt \leftarrow 0$
5 **while** $Q \neq \emptyset$ **do**
6 choisir et supprimer x de Q
7 $cnt \leftarrow cnt + 1$
8 **foreach** $c \in cons(P) \mid x \in scp(c) \wedge stamp[x] > stamp[c]$ **do**
9 **foreach** *variable* $y \in scp(c) \mid y \notin past(P)$ **do**
10 **if** $y \neq x$ *or* $\exists z \in scp(c) \mid z \neq x \wedge stamp[z] > stamp[c]$ **then**
11 **if** *revise*(c, y) **then**
12 **if** $dom(y) = \emptyset$ **then**
13 **return** *false*
14 insert(Q, y)
15 $time \leftarrow time + 1$
16 $stamp[c] \leftarrow time$
17 **if** $cnt \geq threshold$ **then**
18 break
19 **return** *true*

Algorithme 2: insert(Q : ensemble de variables, x : variable)

1 $Q \leftarrow Q \cup \{x\}$
2 $time \leftarrow time + 1$
3 $stamp[x] \leftarrow time$

Algorithme 3: revise(c : contrainte, x : variable)

Résultat : $true$ ssi la révision de (c, x) est effective

1 $nbElements \leftarrow |dom(x)|$
2 **foreach** *value* $a \in dom(x)$ **do**
3 **if** $\neg$*seekSupport*(c, x, a) **then**
4 supprimer a de $dom(x)$
5 **return** $nbElements \neq |dom(x)|$

MAC, qui sera décrit dans la suite. Lors de l'initialisation, nous avons $past(P) = \emptyset$ car il n'y a aucune variable assignée et $\mathsf{GAC}^{arc}(vars(P))$ calcule simplement la GAC-fermeture de P.

Pour établir GAC, les variables sont sélectionnées de manière itérative dans Q (ligne 6). Chaque contrainte c qui porte sur la variable sélectionnée x est prise en compte si $stamp[x] > stamp[c]$. Dans ce cas, chaque variable non assignée $y \neq x$ dans $scp[c]$ est révisée par rapport à la contrainte c. Chaque révision est effectuée par la fonction revise, de l'algorithme 3, qui retourne $true$ si une valeur, au moins, a été supprimée. Si la révision est effective, y est ajoutée à Q. Pour garantir que c est encore GAC-cohérente, nous devons déterminer si l'un des domaines des autres variables que x dans c a été modifié depuis le dernier établissement de GAC sur c. Ceci est pris en charge par la seconde partie de la condition de la ligne 10 de l'algorithme 1. Si la seconde partie de la condition est vérifiée alors x est une variable qui doit être révisée. Si un domaine devient vide, l'algorithme 1 retourne $false$ à la ligne 13. Dans l'algorithme 3, pour chaque valeur a dans $dom(x)$ la fonction seekSupport indique si il existe un support pour (x, a) sur c. De nombreuses implémentations de seekSupport ont été proposées telles que celles utilisées avec (G)AC3, GAC2001 and GAC3^{rm}.

4 Contrôle de la propagation

Dans cet article, nous considérons l'algorithme MAC (Maintaining Arc Consistency), un algorithme de recherche complète considéré comme l'un des plus efficaces pour la résolution générique d'instances CSP. MAC [21] parcourt l'espace de recherche en profondeur d'abord avec retours-arrière et établit la propriété d'arc cohérence (généralisée) après chaque décision prise en cours de recherche. Il construit un arbre de recherche binaire tel que, à chaque noeud interne v, un couple (x, a) est sélectionné, x étant une variable non assignée, et a une valeur appartenant à $dom(x)$. On considère alors deux cas : l'assignation $x = a$ (décision positive) et la réfutation $x \neq a$ (décision négative).

Une variable *passée* est (explicitement) assignée, tandis qu'une variable *future* n'est pas (explicitement) assignée. Pour maintenir GAC au cours de la recherche, on fait appel à $\mathsf{GAC}^{var}(\{x\})$ après chaque décision (positive ou négative) prise sur une variable x. Dans le cas d'une décision positive, nous savons que x appartient désormais aux variables passées. En conséquence, la queue Q ne contient initialement que la variable x.

Par la suite, nous nous référons également à *forward checking* (FC), algorithme de recherche [19, 11] qui maintient une forme partielle d'arc cohérence (généralisée). Au niveau des réseaux de contrainte binaires, lorsqu'une variable x est assignée, seules les variables futures (non assignées) directement connectées à x sont révisées. Le filtrage via FC n'est pas exécuté en phase de pré-traitement (i.e. avant la recherche, lorsqu'aucune variable n'est assignée), ni après une décision négative. Remarquons que, dans le cadre de réseaux de contraintes non binaires, il existe différentes possibilités de généralisation de FC [3].

Nous nous intéressons dans cet article à deux propriétés particulières de la propagation :

- la *longueur* de la propagation, qui correspond au nombre de variables qui ont été extraites de la queue via un appel à la fonction GAC^{var},
- et le *résultat* de la propagation qui correspond à la valeur, $true$ ou $false$, renvoyée par GAC^{var}.

Plus précisément, nous avons étudié l'existence d'une corrélation entre la longueur de la propagation, et son résultat. On observe alors que pour de nombreuses séries d'instances [1], la longueur moyenne de la propagation est significativement plus petite lorsque la fonction GAC^{var} renvoie $false$ que lorsqu'elle renvoie $true$.

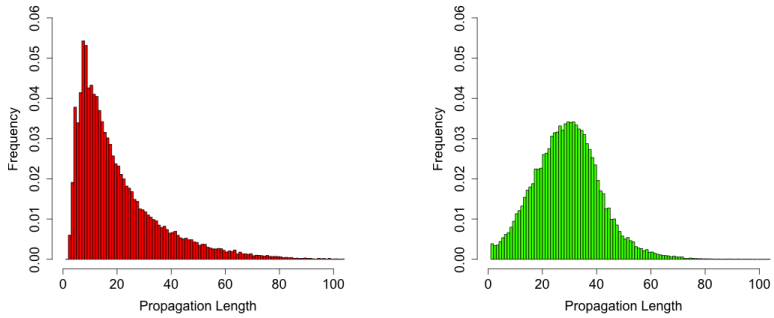
Par exemple, lors de la résolution de l'instance de mots croisés cw-words-vg5-9, MAC (avec l'heuristique *dom/wdeg*) appelle environ 91 000 fois GAC^{var}. 58 000 appels renvoient $false$ et 33 000 renvoient $true$. La distribution précise (en pourcentages) est donnée par la figure 1(a). La longueur moyenne des résultats $false$ (notée avg-false) est de 20.8, tandis qu'elle est de 29.7 pour les résultats $true$ (avg-true). D'autres cas de figure présentent des écarts encore plus prononcés, comme le montrent très clairement les figures 1 et 2. Les valeurs avg-false et avg-true suivantes sont obtenues :

- 4.4 et 24.7 pour l'instance val8-13
- 8.8 et 28.3 pour l'instance langford-3-12
- 9.8 et 12.8 pour l'instance qcp-15-120-5
- 4.2 et 12.6 pour l'instance rand-3-20-20-60-632-18
- 2.7 et 23.9 pour l'instance scens11-f4
- 2.0 et 45.3 pour l'instance lemma-50-9-mod

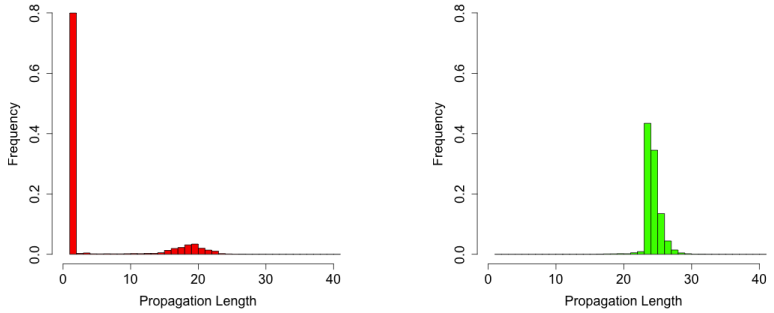
Précisons que chaque résultat présenté ci-dessus est représentatif du fonctionnement général de MAC sur l'ensemble de la série à laquelle appartient l'instance. La valeur de avg-false est en général bien plus petite que celle de avg-true, sauf dans quelques cas, assez rares, où l'inverse est constaté. Par exemple, au niveau de la série pseudo-ii, on obtient de manière assez surprenante 58.9 pour avg-false, et 1.1 pour avg-true sur l'instance pseudo-ii32c1 (voir figure 1(c)).

La mise en évidence d'une corrélation entre la longueur et le résultat de la propagation nous a amenés à développer une nouvelle variante de MAC. Globalement, si nous pouvons déterminer une longueur de propagation à partir de laquelle la fonction GAC^{var} a de fortes probabilités de retourner $true$, pourquoi ne pas l'arrêter dès que cette lon-

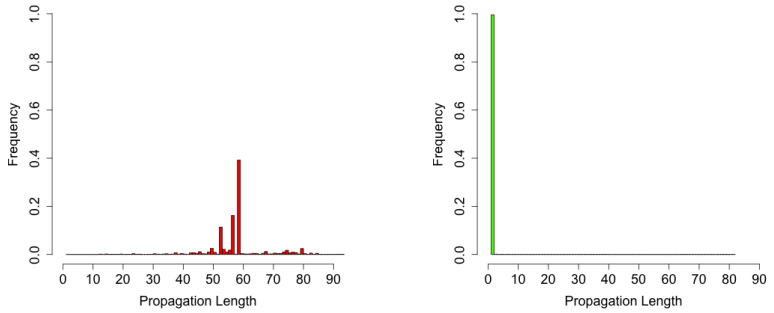
1. Nos expérimentations ont porté sur un nombre conséquent de séries disponibles à `http://www.cril.fr/~lecoutre/benchmarks.html`.



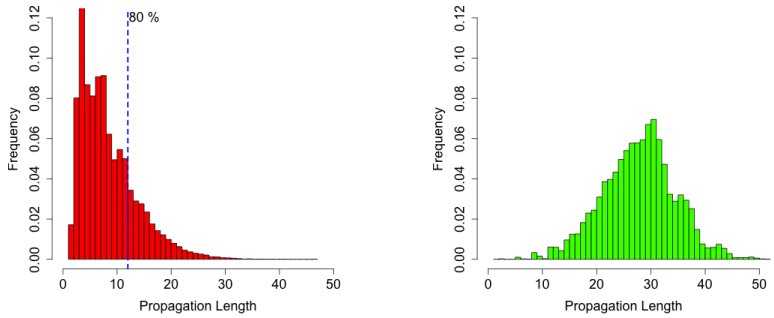
(a) crossword-m1c-words-vg5-9



(b) graph-valiente-8-13



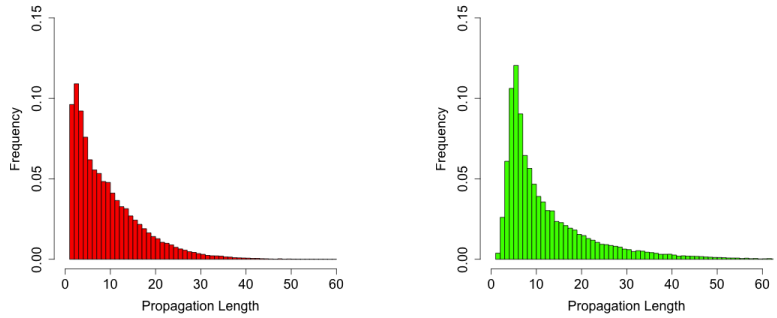
(c) pseudo-ii32c1



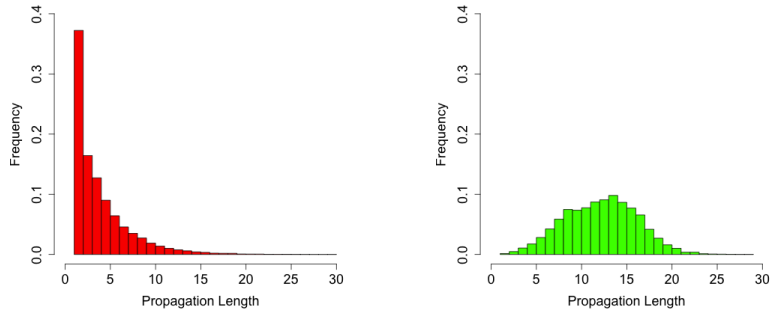
(d) langford-3-12

FIGURE 1 – Corrélation entre longueurs et résultats de propagation : distribution des résultats de propagation $false$ (à gauche) et $true$ (à droite) pour différentes instances.

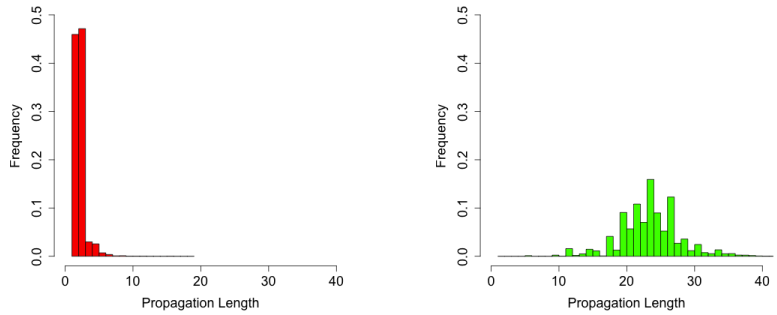
r



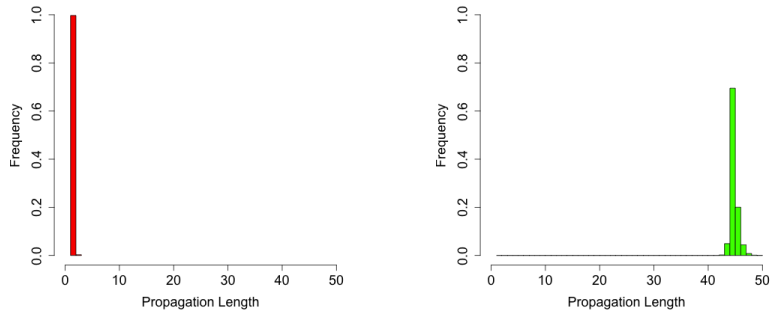
(a) qcp-15-120-5



(b) rand-3-20-20-60-632-18



(c) scens11-f4



(d) lemma-50-9-mod

FIGURE 2 – Corrélation entre longueurs et résultats de propagation : distribution des résultats de propagation $false$ (à gauche) et $true$ (à droite) pour différentes instances.

gueur est atteinte ? Certaines valeurs qui auraient dû être supprimées seront conservées, mais en contrepartie, nous diminuons significativement l'effort nécessaire pour identifier les situations d'échec engendrées par un domaine vide. Nous proposons pour cela MAC^c, qui correspond à l'algorithme 1 dont les lignes grises additionnelles, entraînent les modifications suivantes :

- un compteur cnt est utilisé : initialisé à zéro (ligne 4), il est incrémenté à chaque fois qu'une variable est prise dans Q (ligne 7) ;
- une variable entière $threshold$ est ajoutée : elle correspond à la longueur à laquelle la propagation doit être stoppée ;
- un nouveau test apparaît en fin de boucle (lignes 17-18) pour stopper prématurément la propagation lorsque cela paraît utile.

Remarquons que si $threshold$ vaut 1, MAC^c est équivalent à FC, tandis que si $threshold$ est positionné à l'infini, MAC^c devient équivalent à MAC. En d'autres termes, MAC^c va en général effectuer plus de propagation que FC, mais moins que MAC.

Il nous faut maintenant trouver une bonne valeur de seuil ($threshold$) lors de la résolution d'une instance. A partir de distributions telles que celles des figures 1 et 2, on peut en premier lieu calculer, pour chaque longueur n, le coût moyen $AC(n)$ de détection d'un domaine vide (résultat $false$) lorsque le seuil vaut n. Pour une longueur n, le coût moyen $AC(n)$ est calculé comme suit :

$$\frac{\sum_{i=1}^{n} i \times (F_i + T_i) + \sum_{i=n+1}^{\infty} n \times (F_i + T_i)}{\sum_{i=1}^{n} F_i}$$

où T_i et F_i correspondent respectivement aux nombres de résultats $true$ et $false$ lorsque la longueur est égale à i. Ce coût est exprimé en nombre d'opérations de base effectuées par GAC^{var}, c'est-à-dire en nombre de fois où une variable est extraite de Q. Cette formule signifie que, pour un entier i compris entre 1 et n, le coût est égal à i multiplié par le nombre de fois où la longueur a été égale à i, et pour tout entier i plus grand que n, le coût est égal à n (car la propagation a été stoppée à ce niveau) multiplié par le nombre de fois où la longueur a été égale à i. Ce coût global est alors divisé par le nombre de fois où GAC^{var} a renvoyé $false$ lorsque la longueur était égale à i. Une illustration est donnée par les figures 3, 4 et 5.

Nous pouvons alors contrôler la propagation en positionnant le seuil à la valeur de i qui détermine le coût le plus bas. Dans l'hypothèse où nous connaîtrions à l'avance la distribution, nous obtiendrions, par exemple, des seuils égaux à 1 pour l'instance val8-13, à 12 pour langford-3-12, ou encore à 149 pour pseudo-ii32c1 (voir les figures 1(b), 1(d) et 1(c)). En d'autres termes, MAC^c pourrait identifier 80% des choix conduisant à un domaine vide pour les instances val8-13 et langford-3-12, et 100% pour pseudo-ii32c1 (voir les figures 1(b), 1(d) et 1(c)). Cela signifie également que si de telles valeurs de seuil avaient été fixées dès le début de la résolution, MAC^c se serait comporté de manière similaire à FC sur l'instance val8-13, et de manière similaire à MAC sur l'instance pseudo-ii32c1. En pratique, il n'est pas possible de déterminer ces valeurs à l'avance, aussi nous proposons une procédure dynamique et adaptative de calcul du seuil au cours de la recherche, ce qui produit des variations des valeurs du seuil autour des valeurs théoriques présentées plus haut. Sur nos exemples, MAC^c se comporte alors de façon légèrement différente de FC ou de MAC.

Au niveau de notre implémentation, nous avons utilisé le coût moyen minimum comme valeur de seuil. En fait, le

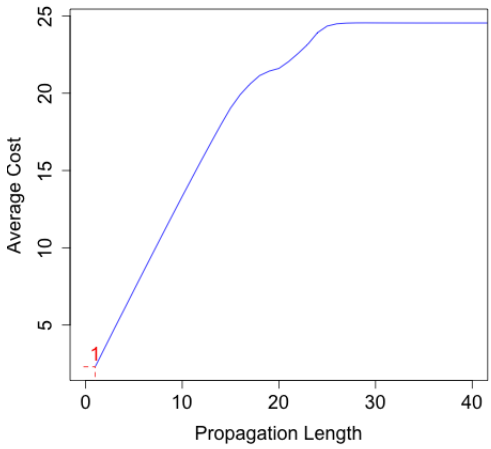


FIGURE 3 – Coût moyen en fonction de la longueur de propagation pour l'instance val8-13.

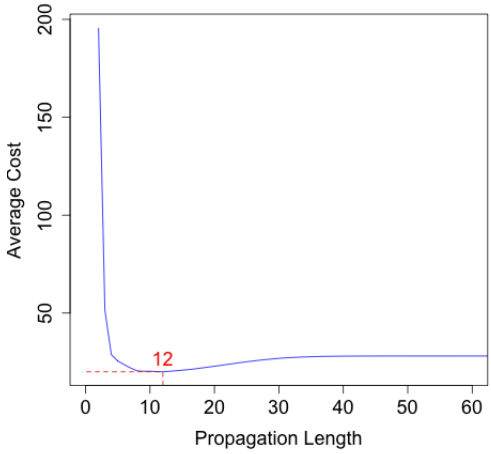


FIGURE 4 – Coût moyen en fonction de la longueur de propagation pour l'instance langford-3-12.

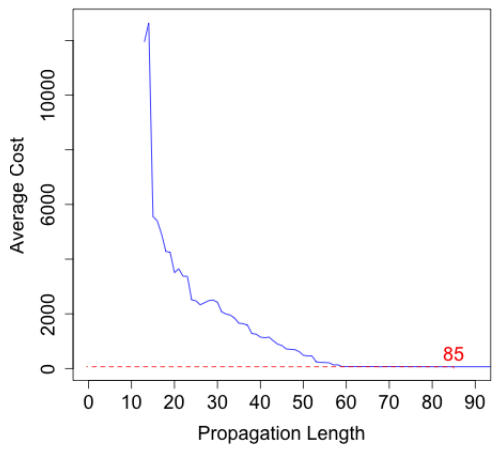


FIGURE 5 – Coût moyen en fonction de la longueur de propagation pour l'instance pseudo-ii32c1.

seuil est recalculé en permanence au cours de la recherche en fonction des résultats précédemment obtenus. Nous calculons le seuil de la façon suivante : nous initialisons la variable *threshold* à partir des s premiers appels à la fonction GAC^{var} (sans contrôle de la propagation). Autrement dit, au début de la recherche, *threshold* est initialisé à l'infini. La longueur et le résultat de ces appels sont enregistrés dans une structure de file (FIFO) de taille s. La valeur de seuil ainsi calculée est alors utilisée pour f appels à GAC^{var}. Le $f+1^{ème}$ appel à GAC^{var} est effectué sans contrôle (comme si *threshold* était positionné à l'infini – ceci n'est pas visible sur l'algorithme). La file va ensuite être mise à jour par simple suppression de la plus ancienne valeur et ajout de la plus récente, ce qui permet de faire évoluer la valeur du seuil. Ce processus est répété cycliquement jusqu'à la fin de la recherche. Pour nos expérimentations, nous avons arbitrairement fixé les valeurs de s à 100, et de f à 10. Les résultats obtenus sont par conséquent préliminaires, et nécessiteront une étude plus poussée de détermination des paramètres en considérant, parmi d'autres pistes, les modèles de séries temporelles (par exemple, les modèles auto-régressifs) utilisés en économétrie [1]. Enfin, remarquons que le fait que *threshold* soit toujours supérieur ou égal à 1 nous assure de la correction de l'algorithme MAC^c : le niveau de filtrage reste au minimum celui de FC.

5 Résultats expérimentaux

Pour montrer l'intérêt pratique du contrôle de la propagation par contraintes à partir d'une analyse de nature statistique concernant la longueur de propagation, nous avons comparé l'efficacité de MAC, MAC^c et FC pour résoudre de nombreuses séries d'instances CSP. Avec l'utilisation de l'heuristique de choix de variable *dom/wdeg* [6] pour guider la recherche, 1338 instances parmi les 2053 (issues d'un nombre important de séries d'instances) ont été résolues par les trois algorithmes MAC, MAC^c and FC, en moins de 1200 secondes chacune. En moyenne, MAC^c est environ 10% plus rapide que MAC et deux fois plus rapide que FC. Ceci est illustré par le tableau 1 où, pour chaque série, le nombre nb_s d'instances résolues par les trois algorithmes dans le temps imparti (1200 secondes) ainsi que le nombre total nb d'instances sont donnés sous la forme (#Inst=nb_s/nb) en-dessous du nom de la série. Les temps CPU moyens affichés dans ce tableau sont calculés à partir des instances nb_s identifiées pour chaque série ; le nombre d'instances résolues par les trois algorithmes est indiqué à la ligne marquée par #résolus. MAC^c est particulièrement efficace sur les instances de colorations de graphes, et obtient d'assez bons résultats sur les séries pret, et qcp. Le tableau 2 fournit des détails sur la résolution

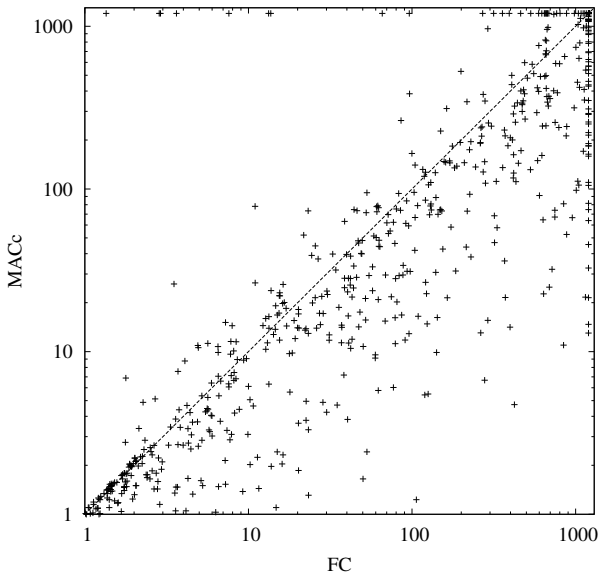
Series		FC	MAC	MAC^c
crosswords-Vg	cpu	92.2	**20.5**	20.7
(#Inst = 200/258)	#résolus	200	214	**217**
graph-coloring	cpu	55.5	58.3	**41.4**
(#Inst = 198/459)	#résolus	210	203	**213**
graph-valiente	cpu	19.9	14.7	**14.2**
(#Inst = 681/793)	#résolus	681	**693**	690
pseudo-ii	cpu	**22.9**	39.1	45.5
(#Inst = 14/41)	#résolus	**31**	16	16
langford	cpu	75.9	**1.06**	1.46
(#Inst = 38/72)	#résolus	38	**47**	**47**
pret	cpu	703	368	**327**
(#Inst = 4/8)	#résolus	4	4	4
qcp	cpu	57.0	48.1	**31.8**
(#Inst = 35/60)	#résolus	**38**	35	**38**
qwh	cpu	45.8	**24.6**	25.3
(#Inst = 30/40)	#résolus	30	30	**31**
rand-3	cpu	190	140	**133**
(#Inst = 120/300)	#résolus	127	**132**	131
scens11	cpu	181	63.6	**59.3**
(#Inst = 9/12)	#résolus	9	**10**	**10**
schurrLemma	cpu	**28.8**	116	50.4
(#Inst = 9/10)	#résolus	9	9	9
All Series	cpu	62.8	36.3	**32.0**
(#Inst = 1338/2053)	#résolus	1387	1393	**1406**

TABLE 1 – Temps moyen (en secondes) pour résoudre les instances des différentes séries (avec un time-out de 1200s pour chaque instance) avec MAC-*wdeg/dom*.

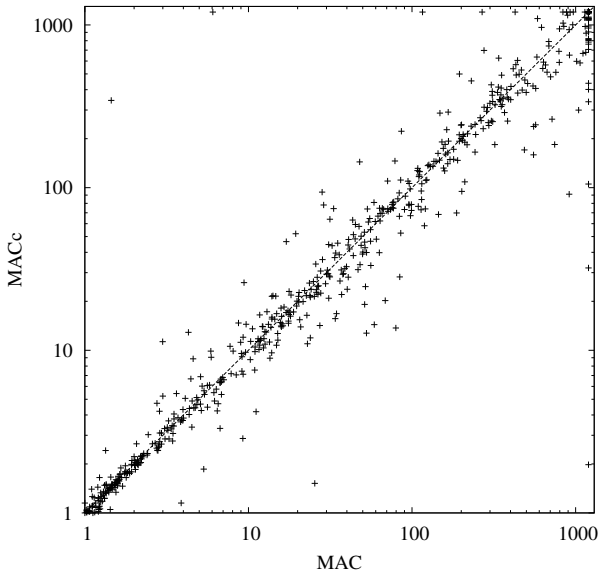
Instances		FC	MAC	MAC^c
cw-words-vg5-9	cpu	308	56.0	**47.0**
	nds	$6,065K$	$127K$	$143K$
2-insertions-5-3	cpu	17.5	**9.7**	**9.7**
	nds	$447K$	$151K$	$163K$
	ccks	$2,149K$	$1,705K$	$1,495K$
david-10	cpu	366	276	**225**
	nds	$9,998K$	$4,439K$	$4,440K$
	ccks	$158M$	$163M$	$148M$
graph-val-8-13	cpu	29.7	27.3	**23.6**
	nds	$826K$	$405K$	$433K$
	ccks	$16M$	$14M$	$11M$
pseudo-ii-32c1	cpu	**7.6**	12.9	14.3
	nds	$120K$	$49,348$	$60,824$
	ccks	$1,107K$	$5,999K$	$6,506K$
langford-3-12	cpu	$1,059$	**13.2**	21.7
	nds	$31M$	$179K$	$345K$
	ccks	$701M$	$4,616K$	$12M$
pret-60-25	cpu	662	354	**323**
	nds	$22M$	$12M$	$10M$
qcp-15-120-5	cpu	62.7	29.9	**21.1**
	nds	$1,742K$	$601K$	$420K$
	ccks	$6,316K$	$7,498K$	$4,617K$
qwh-20-166-0	cpu	157	76.4	**72.6**
	nds	$4,213K$	$1,347K$	$1,348K$
	ccks	$14M$	$17M$	$15M$
rand-3-20-18	cpu	48.6	31.1	**28.6**
	nds	$315K$	$43,952$	$56,085$
scen11-f4	cpu	$1,191$	358	**338**
	nds	$20M$	$3,381K$	$3,718K$
	ccks	$757M$	$261M$	$277M$
lemma-50-9-mod	cpu	**54.0**	202	89.9
	nds	$660K$	$204K$	$204K$
	ccks	$86M$	$706M$	$280M$

TABLE 2 – Résultats détaillés sur diverses instances (time-out de 1200 secondes par instance) avec MAC-*dom/wdeg*.

de diverses instances de la série. Pour chaque instance, le temps CPU total pour la résoudre est donné ainsi que le nombre de noeuds explorés (nds) et le nombre (ccks) de tests de contraintes (sauf pour les contraintes non-binaires en extension où STR2 [13] est utilisé pour appliquer GAC). En général, lorsque MAC et MAC^c explorent à peu près le même arbre de recherche, MAC^c évite de nombreux tests de contraintes, comme on peut le voir pour david-10, qwh-20-166-0, lemma-50-9-mod. Une autre vision des résultats est donné par la figure 6 qui représente des nuages de points permettant une comparaison par paires pour MAC^c opposé à FC et à MAC.



(a) MAC^c est représenté contre FC.



(b) MAC^c est représenté contre MAC.

FIGURE 6 – Comparaison par paires (temps CPU) sur 2053 instances de diverses séries. Le time-out pour résoudre une instance est 20 minutes.

6 Conclusion

Dans cet article, nous avons montré qu'il existe une corrélation intéressante entre la *longueur* et le *résultat* de la propagation de contraintes. Cette observation nous a permis de faire des prévisions raisonnables quant à la capa-

cité de la propagation de contraintes à détecter une incohérence et, par conséquent, nous a conduit à proposer une variante de l'algorithme de recherche MAC lorsque la propagation est stoppée prématurément (i.e. lorsque la probabilité de retour-arrière à partir du noeud courant est faible). Nous croyons que cette corrélation, observée pour la première fois dans la littérature, ouvre de nombreuses autres perspectives pour améliorer l'efficacité de la résolution de contraintes.

Dans un proche avenir, nous projetons de développer des alternatives à notre mode actuel de calcul des valeurs de seuil par exemple, une procédure qui garantisse que x% d'échecs puisse être identifié (où x est l'objectif fixé par l'utilisateur). Nous aimerions également déterminer si certaines caractéristiques des graphes de contraintes telles que la densité, le diamètre, la longueur caractéristique des chemins et/ou le coefficient de regroupement (clustering) [25], peuvent être directement liées à l'efficacité du contrôle de propagation. D'autre part, nous avons l'intention de combiner l'approche probabiliste d'inférence [20] avec notre approche probabiliste de détection d'incohérence, car elles semblent être orthogonales. Enfin, parmi les perspectives que nous avons identifiées, une dernière piste intéressante est l'étude des techniques de shaving s'appuyant sur des tests singletons à coût d'obtention réduit. Cela est prometteur puisque seul le résultat de la propagation est utile pour le shaving. De manière générale, cela pourrait être lié également à la cohérence d'arc paresseuse[22].

Références

[1] *Handbook of Econometrics*. Elsevier, 1983–2007.

[2] C. Bessiere. Constraint propagation. In *Handbook of Constraint Programming*, chapter 3. Elsevier, 2006.

[3] C. Bessiere, P. Meseguer, E.C. Freuder, and J. Larrosa. On Forward Checking for non-binary constraint satisfaction. *Artificial Intelligence*, 141 :205–224, 2002.

[4] C. Bessiere, J.C. Régin, R. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2) :165–185, 2005.

[5] F. Boussemart, F. Hemery, and C. Lecoutre. Revision ordering heuristics for the constraint satisfaction problem. In *Proceedings of CPAI'04 workshop held with CP'04*, pages 29–43, 2004.

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

[7] K. Cheng and R. Yap. An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2) :265–304, 2010.

[8] A. Chmeiss and L. Sais. Constraint satisfaction problems : Backtrack search revisited. In *Proceedings of ICTAI'04*, pages 252–257, 2004.

[9] R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.

[10] I.P. Gent, C. Jefferson, I. Miguel, and P. Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proceedings of AAAI'07*, pages 191–197, 2007.

[11] R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14 :263–313, 1980.

[12] C. Lecoutre. *Constraint networks : techniques and algorithms*. ISTE/Wiley, 2009.

[13] C. Lecoutre. STR2 : Optimized simple tabular reduction for table constraint. *Constraints*, to appear, 2011.

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

[15] C. Lecoutre and R. Szymanek. Generalized arc consistency for positive table constraints. In *Proceedings of CP'06*, pages 284–298, 2006.

[16] O. Lhomme and J.C. Régin. A fast arc consistency algorithm for n-ary constraints. In *Proceedings of AAAI'05*, pages 405–410, 2005.

[17] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1) :99–118, 1977.

[18] A.K. Mackworth. On reading sketch maps. In *Proceedings of IJCAI'77*, pages 598–606, 1977.

[19] J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19 :229–250, 1979.

[20] D. Mehta and M.R.C. van Dongen. Probabilistic consistency boosts MAC and SAC. In *Proceedings of IJCAI'07*, pages 143–148, 2007.

[21] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of CP'94*, pages 10–20, 1994.

[22] T. Schiex, J.C. Régin, C. Gaspin, and G. Verfaillie. Lazy arc consistency. In *Proceedings of AAAI'96*, pages 216–221, 1996.

[23] K. Stergiou. Heuristics for dynamically adapting propagation. In *Proceedings of ECAI'08*, pages 485–489, 2008.

[24] R.J. Wallace and E.C. Freuder. Ordering heuristics for arc consistency algorithms. In *Proceedings of AI/GI/VI'92*, pages 163–169, 1992.

[25] D.J. Watts and S.H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393 :440–442, 1998.