Heuristiques de choix de variables dirigées par les conflits

Frédéric Boussemart **Fred Hemery** **Christophe Lecoutre**
Lakhdar Sais

CRIL-CNRS FRE 2499
rue de l'universite, SP 16
62307 Lens cedex
{boussemart,hemery,lecoutre,sais}@cril.univ-artois.fr

Résumé

Dans ce papier, nous présentons une heuristique de choix de variables dynamique qui dirige la recherche vers les parties difficiles ou inconsistantes des problèmes de satisfaction de contrainte (CSPs). Tout au long du processus de recherche, cette heuristique générique exploite l'expérience des états précédents tandis que les heuristiques traditionnelles ne se basent que sur l'état courant. Plus précisément, cette heuristique réduit le phénomène appelé *thrashing* en choisissant en priorité les variables liées aux contraintes qui conduisent le plus souvent à des situations de conflit. Une telle information est enregistrée en associant un poids à chaque contrainte. Ce poids est incrémenté lorsque la contrainte associée est violée au cours de la recherche. Les nombreuses expérimentations présentées, concernant différents problèmes réels, académiques ou aléatoires, montrent que cette approche, dirigée par les conflits, est la plus efficace.

1 Introduction

Depuis quelques années, de nombreuses améliorations ont été apportées aux algorithmes de recherche arborescente dans le cadre de la résolution des problèmes de satisfaction de contraintes (CSPs). Ces améliorations concernent principalement les heuristiques de choix de variables, les techniques de filtrage, et les analyses de conflit. De nombreux travaux se sont focalisés sur l'amélioration des phases de progression et de régression en introduisant des schémas prospectifs (*look-ahead*) et rétrospectifs (*look-back*) [DF02].

La phase de progression détermine le traitement à réaliser lorsque l'algorithme doit assigner une nouvelle variable. Il faut alors décider de l'assignation à réaliser, des modifications à apporter au problème, et finalement de la présence ou non d'une impasse. La phase de régression détermine le traitement à réaliser lorsque l'algorithme doit effectuer un retour-arrière après avoir rencontré une impasse. Il faut alors décider du saut en arrière à effectuer et, potentiellement, de ce qui peut être appris par cette impasse.

Ces deux paradigmes ne sont pas considérés comme totalement orthogonaux. Plus précisément, on constate théoriquement [CvB01] et expérimentalement [BR96, CvB01] que l'utilisation de techniques prospectives performantes réduit en général l'apport des techniques rétrospectives. Toutefois, nous pensons qu'une certaine forme de collaboration entre techniques prospectives et rétrospectives peut s'avérer intéressante, en particulier lorsque des informations enregistrées au cours de la phase de régression sont exploitées au cours de la phase de progression. En effet, affiner la phase de progression de manière à prévenir au plus tôt le phénomène de thrashing (i.e. la redécouverte des mêmes inconsistances durant la recherche) nécessite une forme d'apprentissage à partir des échecs rencontrés.

Nous proposons ici une approche qui combine avantageusement l'aspect prospectif et rétrospectif des algorithmes de recherche arborescente. Notre objectif principal est de traiter plus efficacement les problèmes structurés, où certaines contraintes apparaissent comme plus importantes que d'autres, et où, par voie de conséquence, certaines parties du problème sont soit inconsistantes, soit difficiles à résoudre. Dans ce but, nous introduisons une heuristique de choix de variables évolutive, $wdeg$, qui dirige systématiquement la recherche vers les parties difficiles ou inconsistantes d'un problème. Notre approche permet d'apprendre (aspect rétrospectif) puis d'exploiter (aspect prospectif) les informations issues des états précédents du processus de recherche alors que les heuristiques traditionnelles, statiques et dynamiques, se limitent respectivement à l'exploitation de l'état initial et courant de la recherche.

Plus précisément, nous proposons une heuristique inspirée par les travaux de [Mor93, SK93, MSG98, BGS99, Tho00, BS00] sur SAT (problème de la satisfiabilité d'une formule propositionnelle) et CSP. Le principe est d'attribuer un poids plus important aux contraintes qui ont été violées au cours des étapes précédentes de la recherche. Ces pondérations sont alors utilisées par $wdeg$ pour guider un algorithme de recherche arborescente. En effet, au cours de la recherche, le poids des contraintes les plus dures devient de plus en plus élevé, ce qui permet de sélectionner les variables apparaissant dans les parties les plus difficiles du CSP. Contrairement aux schémas rétrospectifs sophistiqués (voir, e.g. [DF02]), qui utilisent une technique d'analyse des conflits relativement complexe et des structures de données assez lourdes, notre approche peut être très simplement greffée à un algorithme de recherche arborescente.

Les nombreuses expérimentations, menées sur différentes classes de problèmes réels, académiques ou aléatoires, montrent que cette approche, dirigée par les conflits, est la plus efficace dans une vaste majorité de cas. On observe un gain très important non seulement sur les problèmes structurés contenant une partie inconsistante, mais également sur des problèmes aléatoires (sans noyau inconsistant) et académiques, ce qui confirme la robustesse et l'efficacité de notre approche.

Ce papier est organisé comme suit. Nous rappelons dans un premier temps quelques définitions classiques dans le domaine des CSPs. Après un tour d'horizon des différentes heuristiques classiques de choix de variables, nous décrivons et justifions notre approche. Nous présentons ensuite les résultats d'expérimentations, réalisées sur un large éventail de problèmes. Enfin, avant de conclure, nous situons notre contribution par rapport aux travaux réalisés dans un cadre similaire.

2 Préliminaires

Dans cette section, nous introduisons quelques définitions et notations utilisées dans la suite du papier.

Définition 1 *Un réseau de contraintes est un couple* $(\mathscr{V}, \mathscr{C})$ *où :*
- $\mathscr{V} = \{V_1, \ldots, V_n\}$ *est un ensemble fini de* n *variables tel que chaque variable* V_i *possède un domaine* $dom(V_i)$ *représentant l'ensemble des valeurs pouvant être affectées à* V_i*,*
- $\mathscr{C} = \{C_1, \ldots, C_m\}$ *est un ensemble fini de* m *contraintes tel que chaque contrainte* C_j *correspond à une relation* $rel(C_j)$ *représentant l'ensemble des tuples autorisés pour les variables* $vars(C_j)$ *liées par la contrainte* C_j*.*

Sans perte de généralité, nous considérerons que tout ensemble de variables $vars(C)$ associé à une contrainte C est ordonné. Ainsi, il est possible d'obtenir la position $pos(V, C)$ d'une variable V dans $vars(C)$. Un tuple t est dit autorisé par une contrainte C lorsque t appartient à $rel(C)$, et est dit être un support d'un couple (V, v) dans C lorsque t est autorisé par C et est tel que $t[pos(V, C)] = v$.

Nous dirons qu'une contrainte C lie (ou implique) une variable V si et seulement si V appartient à $vars(C)$. L'arité d'une contrainte C est le nombre de variables liées par celle-ci, i.e., le nombre de variables dans $vars(C)$. Une contrainte binaire lie deux variables. Le degré d'une variable V est le nombre de contraintes liant celle-ci. Deux variables V_i et V_j sont dites voisines ssi il existe une contrainte C telle que $\{V_i, V_j\} \in vars(C)$. Nous notons $\Gamma(V_i)$ l'ensemble des variables voisines de V_i.

Une assignation (également appelée instanciation) de variable est l'affectation d'une valeur à une variable. Une solution est l'assignation de toutes les variables du réseau de telle sorte que chaque contrainte soit satisfaite. Une solution garantit l'existence d'un support dans chaque contrainte. Déterminer si un tuple est support est appelé test de consistance.

Le problème (général) de satisfaction de contraintes est la tâche de trouver une solution à un réseau de contraintes, également appelé instance CSP. Ce problème est NP-complet. Une instance CSP est dite satisfiable si et seulement si le réseau auquel elle correspond admet au moins une solution (est dite insatisfiable sinon). Résoudre une instance CSP consiste soit à déterminer son insatisfiabilité, soit à trouver une solution (ou plus). Par abus de langage, CSP et instance CSP sont parfois confondus.

Au niveau de la résolution des CSPs, les méthodes de recherche complètes utilisent, en règle générale, un algorithme de recherche en profondeur d'abord avec gestion de retours-arrières, où à chaque étape de la recherche, une assignation de variable est effectuée suivie par un processus de filtrage appelé propagation de contraintes. Dans la plupart des cas, les algorithmes de propagation de contraintes qui sont basés sur des propriétés des réseaux de contraintes telles que la consistance d'arc, éliminent des valeurs qui ne peuvent apparaître dans aucune solution. L'algorithme de maintient de la propriété de consistance d'arc pendant la phase de propagation de contraintes est appelé MAC ([SF94]) ou encore *really full look-ahead* ([Nad88]).

Définition 2 *Soient* P *une instance CSP,* C *une contrainte de* P*,* V *une variable de* $vars(C)$ *et* v *une valeur de* $dom(V)$*.* (V, v) *est dit consistant par rapport à* C *ssi il existe un support de* (V, v) *dans* C*.* C *est dit arc-consistant ssi pour toute variable* V *de* $vars(C)$ *et pour toute valeur* v *de* $dom(V)$*,* (V, v) *est consistant par rapport à* C*.* P *est arc-consistant ssi toute contrainte de* P *est arc-consistante.*

3 Pondération des contraintes à partir des conflits

Dans la première partie de cette section, nous décrivons brièvement l'ensemble des heuristiques de choix de variables les plus courantes. Nous montrons ensuite, sur un exemple, comment il est possible d'isoler les parties difficiles ou inconsistantes d'un CSP en comptabilisant les violations de contrainte observées au cours de la recherche d'une solution. Enfin, nous proposons une heuristique originale basée sur une pondération dynamique des contraintes pour orienter la recherche vers les parties les plus difficiles d'un CSP.

3.1 Vue d'ensemble des heuristiques de choix de variable

L'ordre dans lequel les variables sont assignées par un algorithme de recherche avec retour-arrière est considéré depuis longtemps comme crucial. En effet, l'application de différentes heuristiques de choix de variables pour résoudre une même instance peut conduire à des résultats extrêmement variés en terme d'efficacité. Plus encore, [GSCK00] ont montré que la simple introduction d'un processus aléatoire au niveau d'une heuristique de choix de variables permet d'observer de grands écarts de performance. En fait, une heuristique idéale consisterait à sélectionner en priorité un *backdoor* ([WGS03]), c'est à dire un ensemble de variables qui, une fois assignées, rendent le problème facile à résoudre.

Les heuristiques généralement utilisées peuvent être classées en plusieurs catégories. La première correspond aux heuristiques statiques (SVOs pour *Static Variable Ordering*) qui conservent le même ordre de priorité tout au long de la recherche, ordre établit à partir de la structure du problème dans son état initial. Les heuristiques les plus courantes sont les suivantes :

– $lexico$: les variables sont ordonnées lexicographiquement,
– deg : les variables sont ordonnées de manière décroissante en fonction de leur degré initial [DM89],
– $ddeg$: les variables sont ordonnées de manière décroissante en fonction de leur degré courant (dit dynamique),
– $width$: les variables sont ordonnées de manière à limiter la largeur du graphe de contraintes [Fre82].

La seconde catégorie correspond aux heuristiques dynamiques (DVOs pour *Dynamic Variable Ordering*) qui exploitent différentes informations sur l'état courant du problème. Une heuristique bien connue, dom [HE80], consiste à ordonner les variables suivant la taille courante de leur domaine. [HE80] justifie l'utilisation de cette heuristique par le principe du *fail-first* : *«To succeed, try first where you are most likely to fail»*. En combinant la taille courante du domaine et le degré, on obtient les heuristiques dom/deg [BR96] et $dom/ddeg$ [BR96, SG98] qui peuvent très nettement améliorer les performances de la recherche. D'autres heuristiques ont été proposées par [GMP$^+$96, HH00], mais bien que conceptuellement élégantes, ces dernières entraînent des calculs supplémentaires importants. De plus, elles n'ont été testées que sur des problèmes aléatoires, ce qui rend leur intérêt pratique difficile à estimer.

D'autre part, on observe que l'ordre généré par ces différentes heuristiques fait apparaître des sous-ensembles de variables équivalentes (*tie*). Nous considérerons ici que $lexico$ est implicitement utilisée pour départager ces variables. Par exemple, dom correspond en fait à $dom \oplus lexico$, qui sélectionne, parmi toutes les variables dont le domaine est le plus petit, celle qui apparaît la première dans l'ordre lexicographique. D'autres heuristiques peuvent être explicitement indiquées pour départager les variables jugées équivalentes (cependant, en dernier recours, $lexico$ sera toujours implicitement utilisée). Parmi les heuristiques composées de cette façon, on peut par exemple citer $dom \oplus deg$ [FD95], $dom \oplus ddeg$ [Bre79, Smi99] et $BZ3$ [Smi99].

Plus récemment, [BCS01] ont proposé une généralisation (notée ici $mDVO$) de l'ensemble des heuristiques de choix de variables existantes, en considérant que chaque variable peut être évaluée en fonction de son voisinage. Plus précisément, si on considère une heuristique h basée sur une fonction d'évaluation des variables α_h, on peut définir à l'aide d'un opérateur $\odot$, une généralisation $h_d^{\odot}$ de l'heuristique h à une distance de voisinage égale à d. Pour une distance égale à 1, la fonction $\alpha_{h,1}^{\odot}$ est définie de la manière suivante pour chaque variable X_i :

$$\alpha_{h,1}^{\odot}(X_i) = \frac{\sum_{X_j \in \Gamma(X_i)}(\alpha_h(X_i) \odot \alpha_h(X_j))}{|\Gamma(X_i)|^2}$$

Par exemple, si $h = dom$ alors $\alpha_h(X) = \alpha_{dom}(X) = |dom(X)|$ et la fonction d'évaluation qui correspond à $dom_1^{\times}$ devient alors :

$$\alpha_{dom,1}^{\times}(X_i) = \frac{|dom(X_i)| \times \sum_{X_j \in \Gamma(X_i)}|dom(X_j)|}{|\Gamma(X_i)|^2}$$

Enfin, de nombreuses expérimentations ont été réalisées pour comparer l'impact des différentes heuristiques de choix de variables. Bien qu'aucune ne surclasse réellement les autres, il est généralement admis que les heuristiques dynamiques telles que dom, $dom \oplus ddeg$ et $dom/ddeg$ sont les plus efficaces. La généralisation de dom et de $dom/ddeg$ au voisinage semble également intéressante.

3.2 Identification des sous-problèmes inconsistants

Dans cette section, nous montrons l'avantage de comptabiliser les violations de contraintes pour isoler les parties difficiles ou inconsistantes d'un CSP. Pour donner une idée de l'intérêt de notre approche, considérons ici un exemple construit par fusion de deux problèmes, l'un étant consistant et l'autre non.

Le premier problème est celui bien connu des reines, qui consiste à placer n reines sur un échiquier de taille $n \times n$ de sorte qu'elles ne soient pas mutuellement en prise. Le nombre de solutions de ce problème facile augmente avec n. Dans une modélisation classique de ce problème, chaque reine est placée sur une ligne, et se trouve représentée par une variable dont le domaine comprend exactement n valeurs (les différentes colonnes de l'échiquier). On définit alors une contrainte pour chaque couple de variable de façon à garantir que les reines ne peuvent pas être placée sur la même colonne ou sur la même diagonale.

Le second problème (également facile) est celui des cavaliers. Celui-ci consiste à placer m cavaliers sur un échiquier de taille $n \times n$ de sorte que l'ensemble des cavaliers forme un cycle (en considérant un lien entre deux cavaliers lorsqu'ils sont placés à une distance d'un saut). Ce problème est insatisfiable lorsque m est impair. Pour le modéliser, on représente chaque cavalier par une variable dont le domaine, de taille n^2 correspond aux différentes cases de l'échiquier. Les contraintes sont définies d'une part pour interdire le placement de deux cavaliers sur une même case, et d'autre part pour garantir que tout cavalier peut atteindre le cavalier suivant (modulo m).

En fusionnant les problèmes des reines et des cavaliers (en nombre impair) de manière indépendante (i.e. sans interaction entre les reines et les cavaliers), on observe, au cours de la résolution, un phénomène appelé *thrashing*. Du fait de la petite taille de leur domaine, les variables représentant les reines sont sélectionnées en priorité par les heuristiques classiques. Le sous-problème (insatisfiable) des cavaliers est alors systématiquement redécouvert pour chaque solution du sous-problème des reines.

Pour mettre en évidence ce phénomène, nous avons exécuté l'algorithme MAC pour résoudre une instance de ce problème en utilisant successivement chacune des heuristiques citées plus haut. Nous avons alors comptabilisé pour chaque contrainte, le nombre de fois où elle a été violée au cours de la recherche.

Nous nous sommes intéressés aux valeurs maximales des compteurs associés d'une part aux contraintes portant sur les reines, et d'autre part à celles portant sur les cavaliers. La figure 1

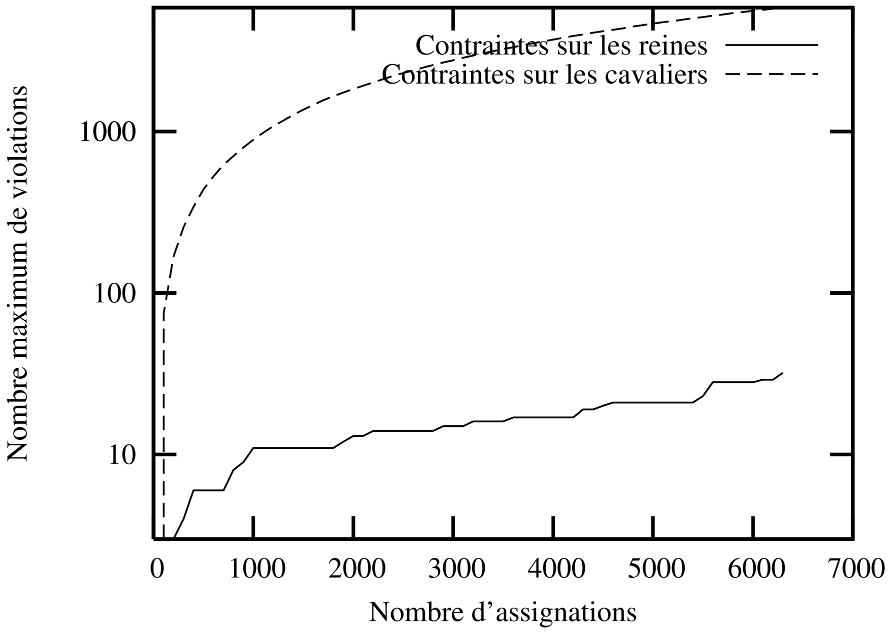

FIG. 1 – Évolution du nombre de contraintes violées pour 5 cavaliers et 8 reines

montre l'évolution de ces deux valeurs, pour l'heuristique[1] $\frac{dom}{ddeg}$, en fonction du nombre d'assignations réalisées pour prouver l'inconsistance d'une instance portant sur 8 reines et 5 cavaliers. Il apparaît clairement que les contraintes portant sur les cavaliers sont bien plus souvent violées que celles portant sur les reines. En d'autres termes, ces compteurs mettent en évidence le sous-problème (inconsistant) des cavaliers.

Cette courbe illustre et confirme que le nombre de violations des contraintes pendant la recherche est une information importante, qui peut être utilisée pour localiser la partie inconsistante (ou difficile) d'un CSP. Nous présentons dans la section qui suit une nouvelle heuristique de choix de variable qui exploite cette information pour diriger la recherche.

3.3 Heuristiques de choix de variables dirigées par les conflits

Comme cela a été indiqué précédemment, les heuristiques de choix de variables traditionnelles exploitent certaines informations concernant l'état courant de la recherche telles que la taille courante des domaines ou le degré courant (appelé également dynamique) des variables. Cette approche apparaît limitée, puisqu'aucune information concernant les états passés de la recherche n'est exploitée. Nous proposons d'enregistrer une information de ce type en associant un compteur, appelé $weight$, à chaque contrainte du problème, compteur qui sera mis à jour pendant la recherche chaque fois qu'un conflit (impasse) apparaît.

Comme les algorithmes de recherche arborescente tels que FC [HE80] ou MAC [SF94] nécessitent des révisions successives de variables de manière à éliminer les valeurs devenues inconsistantes par rapport à l'état courant, il suffit d'introduire un test après chaque révision

[1] Le même comportement peut être observé pour d'autres heuristiques telles que dom, $dom \oplus ddeg$, ...

Algorithme 1 filtrer(Q : Ensemble d'arcs) : booleen

while $Q \neq \emptyset$ **do**
  sélectionner et éliminer (C, V) de Q
  if $reviser(C, V)$ **then**
    if $dom(V) = \emptyset$ **then**
      weight[C]++ *// incrémentation du compteur de la contrainte*
      retourner ECHEC
    else
      mettre à jour Q *// en fonction de l'algorithme de filtrage*
    end if
  end if
end while
retourner SUCCES

pour déterminer si un conflit est apparu. En fait, cette vision correspond à l'utilisation d'un filtrage à gros grain sur la base d'un algorithme tel que AC3 [Mac77], AC2001/3.1 [BR01, ZY01], AC3$_d$ [vD02], AC3.2 [LBH03b] et AC3.3 [LBH03b]. Néanmoins, il est facile d'appliquer le même principe à l'utilisation d'un filtrage à grain fin tel que AC4 [MH86], AC6 [Bes94] et AC7 [BFR99]).

Considérant donc l'utilisation d'un filtrage à gros grain (et une variante orienté-arc [LBH03a] pour la présentation), un ensemble noté Q est utilisé de manière à mémoriser tous les arcs (i.e., couples composés d'une contrainte et d'une variable) qui doivent être révisés ; l'objectif de la révision d'un arc (C, V) étant d'éliminer les valeurs inconsistantes de $dom(V)$ par rapport à la contrainte C.

Lors du déroulement d'un algorithme tel que FC ou MAC, il est nécessaire d'effectuer de façon récurrente le filtrage de certains arcs. Pour cela, la fonction $filtrer$ est appelée (Algorithme 1) avec comme donnée un ensemble d'arcs à réviser (l'initialisation de cet ensemble est spécifique à FC et MAC). Chaque arc est révisé à tour de rôle, et lorsqu'une révision est effective (au moins une valeur a été éliminée), il est nécessaire de contrôler qu'une impasse n'a pas été rencontrée. Si c'est le cas, le compteur associé à la contrainte est incrémenté et le filtrage se termine sur un échec (nécessitant ainsi un retour-arrière), sinon l'ensemble Q est mis à jour (pour FC, rien à faire et pour MAC, $Q \leftarrow Q \cup \{(C', V') \mid V \in vars(C') \wedge V' \in vars(C') \wedge V \neq V' \wedge C \neq C'\}$.

Algorithme 2 reviser(C : Contrainte, V : Variable) : booleen

nbElements $\leftarrow \mid dom(V) \mid$
for chaque $v \in dom(V)$ **do**
  if $rechercherSupport(C, V, v) = false$ **then**
    remove a from $dom(X)$
  end if
end for
retourner nbElements $\neq \mid dom(V) \mid$

Une révision est réalisée par un appel à la fonction $reviser$ (Algorithme 2), et implique d'éliminer les valeurs de $dom(V)$ qui sont devenues inconsistantes par rapport à C. En fonction de l'implémentation de la fonction $rechercherSupport(C, V, v)$ qui détermine si oui ou non un support de (V, v) dans C existe, on obtient différents algorithmes à gros grain.

En utilisant ces compteurs, il est possible de définir une nouvelle heuristique de choix de variables, notée $wdeg$, qui donne une évaluation $\alpha_{wdeg}(V_i)$, appelée degré pondéré, de toute variable V_i comme suit :

$$\alpha_{wdeg}(V_i) = \sum_{V_i \in vars(C) \wedge |FutVars(C)|>1}(weight[C])$$

où $FutVars(C)$ représente le nombre de variables futures (i.e. variables non assignées) dans $vars(C)$.

Ainsi, le degré pondéré d'une variable V_i correspond à la somme des poids des contraintes liant V_i et au moins une autre variable future. Intuitivement, les parties inconsistantes ou simplement difficiles d'un CSP devraient être explorées en priorité par l'algorithme de recherche en sélectionnant d'abord les variables avec le plus grand degré pondéré, respectant ainsi le principe *fail-first*.

Il est important de noter que cette nouvelle heuristique est liée à $ddeg$ puisque seules les contraintes impliquant une variable future donnée et au moins une autre variable future sont prises en compte. En fait, lorsque tous les compteurs $weight$ sont fixés à 1, $wdeg$ et $ddeg$ deviennent équivalents. Aussi, de manière à bénéficier, dés le début de la recherche, du degré des variables, tous les compteurs $weight$ sont initialisés à 1.

Finalement, en combinant degré pondéré des variables et taille courante des domaines, on obtient $dom/wdeg$, une heuristique qui consiste à sélectionner en premier lieu la variable avec le plus petit rapport : taille courante du domaine de la variable sur degré pondéré courant de la variable. Dans le reste de l'article, $wdeg$ et $dom/wdeg$ seront appelées heuristiques (de choix de variables) dirigées par les conflits.

4 Résultats comparatifs

Pour montrer l'intérêt pratique des heuristiques dirigées par les conflits qui ont été présentées dans ce papier, nous avons implémenté celles-ci sur la plate-forme *AbsCon* [LBH04a], et réalisé différentes expérimentations (sur un PC Pentium IV 2,4GHz 512Mo sous Linux) sur différentes classes de problèmes réels, académiques et aléatoires. Nous avons évalué les différentes heuristiques selon le nombre de tests de consistance (#ccks), le nombre d'assignations (#asgs) et le temps cpu en secondes (cpu). Également, pour certains problèmes impliquant plusieurs instances, le nombre d'instances résolues (#slvs) en deçà d'un temps limite fixé ici à 1 heure est indiqué et les performances sont données en moyenne. Il faut noter que lorsque certaines instances de tels problèmes n'ont pas été résolues, les mesures de performance représentent des minorations des valeurs réelles.

Nous avons utilisé un algorithme MAC (appelé GAC pour les problèmes non binaires) qui intègre l'algorithme de consistance d'arc à gros grain $AC3.2$ [LBH03b] avec un schéma de propagation orienté variable et l'heuristique de choix de révision dom^v qui ordonne les variables dans la queue de propagation par taille croissante des domaines courants (pour plus de détails voir [LBH03a]).

4.1 Exemple illustratif

Notre objectif est ici de montrer, à partir de l'exemple illustratif défini section 3.2, comment l'utilisation de $wdeg$ permet d'améliorer très nettement les performances de MAC par rapport à celles obtenues avec $\frac{dom}{ddeg}$.

n	$instances$		$wdeg$	$\frac{dom}{wdeg}$	$\frac{dom}{ddeg}$	$dom \oplus ddeg$
8	K_5	cpu	0.16	0.15	0.12	0.13
		#ccks	0.089M	0.083M	0.076M	0.076M
		#asgs	0.14K	0.09K	0.06K	0.06K
	$K_5 \oplus Q_8$	cpu	0.34	0.33	3.60	3.56
		#ccks	0.373M	0.378M	6.719M	6.719M
		#asgs	0.56K	0.38K	6.37K	6.46K
	$K_5 \otimes Q_8$	cpu	0.38	0.63	3.04	3.03
		#ccks	0.407M	0.776M	5.164M	5.164M
		#asgs	0.53K	0.78K	5.72K	5.72K
10	K_5	cpu	0.22	0.25	0.22	0.23
		#ccks	0.214M	0.201M	0.185M	0.185M
		#asgs	0.21K	0.14K	0.09K	0.09K
	$K_5 \oplus Q_{10}$	cpu	0.77	0.73	65.65	64.19
		#ccks	1.151M	1.099M	130.128M	130.128M
		#asgs	1.10K	0.70K	78.61K	79.33K
	$K_5 \otimes Q_{10}$	cpu	0.96	1.19	56.10	56.07
		#ccks	1.311M	1.830M	106.647M	106.647M
		#asgs	1.07K	1.16K	72.04K	72.04K
12	K_5	cpu	0.36	0.37	0.32	0.29
		#ccks	0.436M	0.413M	0.376M	0.376M
		#asgs	0.31K	0.21K	0.14K	0.14K
	$K_5 \oplus Q_{12}$	cpu	1.74	3.89	2,610.74	2,571.82
		#ccks	2.863M	7.325M	5,188.754M	5,188.754M
		#asgs	1.83K	3.26K	2,174.29K	2,188.49K
	$K_5 \otimes Q_{12}$	cpu	2.36	6.91	2,342.88	2,246.18
		#ccks	3.668M	12.464M	4,356.884M	4,356.884M
		#asgs	2.08K	5.55K	2,017.28K	2,017.28K

TAB. 1 – Instances des reines+cavaliers

Le tableau 1 présente les résultats obtenus sur trois échiquiers de taille différente ($n = 8$, $n = 10$ et $n = 12$) pour trois problèmes différents :

- K_5 : le problème des 5 cavaliers défini section 3.2,
- $K_5 \oplus Q_n$: les problème des 5 cavaliers et des n reines fusionnés de manière indépendante,
- $K_5 \otimes Q_n$: les problèmes des 5 cavaliers et des n reines fusionnés avec interaction (reines et cavaliers ne peuvent pas occuper la même case).

Ces résultats permettent clairement d'observer le phénomène de *thrashing* qui apparaît avec l'heuristique $\frac{dom}{ddeg}$: le nombre d'assignations (resp. de tests de consistance) effectuées pour résoudre les instances $K_5 \oplus Q_n$ ou $K_5 \otimes Q_n$ est approximativement égal au produit du nombre d'assignations (resp. de tests de consistance) effectuées pour résoudre K_5 par le nombre de solutions de l'instance du problème des reines (92 pour n=8, 1024 pour n=10 et 14200 pour n=12). Ce phénomène n'apparaît pas avec les heuristiques dirigées par les conflits. En effet, après avoir trouvé un nombre limité de solutions au problème des reines, le poids des contraintes concernant les cavaliers devient plus important. Les variables qui représentent les cavaliers sont alors sélectionnées en priorité, et le phénomène de *thrashing* disparaît.

4.2 Expérimentations

Dans cette section, nous présentons les expérimentations réalisées sur trois catégories d'instances, à savoir des instances issues de problèmes réels, académiques et aléatoires.

Au niveau des instances réelles issues de l'archive RLFAP (Radio Link Frequency Assignment Problem), nous adoptons l'approche de [BCS01] pour générer des instances difficiles en supprimant certaines contraintes et/ou fréquences. Par exemple, $scen07$-$w1$-$f4$ correspond à l'instance $scen07$ pour laquelle les contraintes de poids supérieur à 1 et les 4 fréquences les plus hautes ont été supprimées. Le tableau 2 présente les résultats obtenus pour une sélection

instances		$wdeg$	$\frac{dom}{wdeg}$	$\frac{dom}{ddeg}$	$mDVO$
$scen11$	cpu	16.28	3.64	93.97	21.05
(sat)	#asgs	2.69K	0.91K	31.81K	8.31K
$scen02$-$f25$	cpu	2.67	1.99	$> 2h$	292.27
(unsat)	#asgs	0.66K	0.89K	-	123.95K
$scen03$-$f10$	cpu	2.06	1.7	$> 2h$	1.8
(sat)	#asgs	0.68K	0.77K	-	0.42K
$scen03$-$f11$	cpu	4.63	2.93	$> 2h$	16.39
(unsat)	#asgs	0.75K	0.8K	-	1.09K
$scen06$-$w2$	cpu	0.31	0.93	$> 2h$	0.25
(unsat)	#asgs	0.02K	0.74K	-	0.01K
$scen07$-$w1$-$f4$	cpu	0.35	0.3	0.44	0.49
(sat)	#asgs	0.44K	0.48K	0.75K	0.76K
$scen07$-$w1$-$f5$	cpu	0.83	0.36	$> 2h$	$> 2h$
(unsat)	#asgs	0.44K	0.25K	-	-
$graph08$-$f10$	cpu	128.85	15.86	$> 2h$	$> 2h$
(sat)	#asgs	29.64K	5.77K	-	-
$graph08$-$f11$	cpu	2.32	17.71	$> 2h$	$> 2h$
(unsat)	#asgs	0.24K	5.07K	-	-
$graph14$-$f27$	cpu	185.61	5.25	$> 2h$	$> 2h$
(sat)	#asgs	39.84K	1.62K	-	-
$graph14$-$f28$	cpu	30	27.79	$> 2h$	$> 2h$
(unsat)	#asgs	4.6K	7.85K	-	-

TAB. 2 – Instances RLFAP

d'instances difficiles. Notons que dans la colonne de droite, nous donnons ici le meilleur résultat obtenu à partir de quatre variantes de l'heuristique $mDVO$: $\alpha^{\times}_{dom,1}$, $\alpha^{\times}_{\frac{dom}{ddeg},1}$,$\alpha^{+}_{dom,1}$ et $\alpha^{+}_{\frac{dom}{ddeg},1}$.

Le bon comportement des heuristiques dirigées par les conflits apparaît ici clairement : elles ont permis de résoudre les différentes instances en quelques secondes[2], alors que $\frac{dom}{ddeg}$ ne permet d'en résoudre que 2 et $mDVO$ 6, la limite de temps ayant été fixée à 2 heures par instance.

Nous avons ensuite testé certains problèmes réalistes de surveillance radar, proposés par le *Swedish Institute of Computer Science (SICS)*. Le problème est d'ajuster la portée du signal (de 0 à 3) d'un nombre donné de radars, placés aléatoirement sur une grille, par rapport à 6 secteurs géographiques. Chaque cellule du territoire doit être couverte par exactement 3 radars, à l'exception de quelques cellules particulières ("ennemies") qui ne doivent pas du tout être couvertes.

Nous avons artificiellement généré 3 jeux de 50 instances définies comme suit sur un territoire de 8×8 cellules :

– rs-30-0 : instances sous-contraintes comprenant 30 radars et aucune cellule particulière,
– rs-24-12-ac : instances sur-contraintes (mais arc-consistantes) comprenant 24 radars et 12 cellules particulières,
– rs-24-2 : instances se situant au point de transition comprenant 24 radars et 2 cellules particulières.

Chaque jeu est noté rs-i-j, i représentant le nombre de radars et j le nombre de cellules particulières. On peut observer sur le tableau 3 les gains énormes obtenus par l'utilisation des heuristiques dirigées par les conflits : avec 1 000 à 10 000 fois plus d'assignations, l'utilisation d'heuristiques classiques n'a permis de résoudre qu'une partie des 50 instances testées, dans la limite de temps fixée à 2 heures par instance.

La deuxième série d'expérimentations porte sur les instances académiques suivantes :

– deux instances du problème de coloration d'échiquier[BLW89], notées cc-7-2 and cc-7-3,

[2] Les seules instances RLFAP modifiées qui n'ont pas pu être résolues avec MAC-$wdeg$ en moins de 2 heures sont les instances $scen11$-$f1$, $scen11$-$f2$ et $scen11$-$f3$

instances		$wdeg$	$\frac{dom}{wdeg}$	$\frac{dom}{ddeg}$	$dom \oplus ddeg$
rs-30-0	cpu	0.05	0.09	623.04	2,319.99
	#asgs	0.2K	0.23K	1,656.95K	12,538.74K
(50/50 sat)	#slvs	50	50	46	35
rs-24-12-ac	cpu	3.78	7.86	2,020.06	2,336.01
	#asgs	1.21K	1.68K	722.36K	641.92K
(0/50 sat)	#slvs	50	50	36	35
rs-24-2	cpu	3.86	3.24	2,030.73	2,545.44
	#asgs	0.92K	0.59K	4,027.56K	7,805.41K
(26/50 sat)	#slvs	50	50	37	33

TAB. 3 – Instances de surveillance radar

comprenant des contraintes d'arité 4,

– deux instances du problème de la règle de *Golomb*[3], notées gr-44-9 et gr-44-10, comprenant des contraintes binaires et ternaires.
– deux instances du problème des nombres premiers attaqués par une reine[4], notées qa-5 et qa-6, composées uniquement de contraintes binaires.

instances		$wdeg$	$\frac{dom}{wdeg}$	$\frac{dom}{ddeg}$	$mDVO$
cc-7-2	cpu	1.87	2.38	10.99	4.34
(unsat)	#asgs	2.23K	5.26K	30.19K	14.78K
cc-7-3	cpu	214.82	39.86	292.28	322.64
(sat)	#asgs	387.3K	104.38K	714.1K	780.56K
gr-44-9	cpu	31.57	60.54	37	47.91
(sat)	#asgs	5.39K	14.8K	11.74K	11.73K
gr-44-10	cpu	708.42	167.79	1,356.64	583.23
(unsat)	#asgs	75.56K	22.38K	202.07K	44.28K
qa-5	cpu	4.50	2.37	70.50	24.97
(sat)	#asgs	7.90K	4.79K	318.60K	100.81K
qa-6	cpu	88.73	82.46	5490.86	339.73
(sat)	#asgs	62.74K	74.58K	7,703.44K	523.14K

TAB. 4 – Instances académiques

Même si les écarts sont ici moins importants, le tableau 4, montre que l'utilisation d'heuristiques dirigées par les conflits reste en général l'approche la plus performante.

La troisième et dernière série de tests concerne les problèmes aléatoires. Pour cela, nous avons considéré les classes d'instances suivantes :

– deux classes de 100 instances satisfiables *balanced Quasigroup With Holes (bQWH)* [GS02] d'ordre 15 avec 106 trous, et d'ordre 18 avec 141 trous,
– trois classes de 100 instances de problèmes aléatoires binaires notés $< n, d, m, t >$ où n représente le nombre de variables, d la taille uniforme des domaines, m le nombre de contraintes binaires, et t la dureté de ces contraintes.

Sur le tableau 5, on remarque que pour ces deux types de problèmes aléatoires, plus la taille du problème est importante, plus l'écart entre les heuristiques dirigées par les conflits (et plus particulièrement $\frac{dom}{wdeg}$) et les heuristiques classiques augmente.

[3]Voir problem006 sur `http://4c.ucc.ie/~tw/csplib/`
[4]Voir problem029 sur `http://4c.ucc.ie/~tw/csplib/`

instances		$wdeg$	$\frac{dom}{wdeg}$	$\frac{dom}{ddeg}$	$dom \bigoplus ddeg$
$bqwh$	cpu	1.06	0.5	4.85	4.94
$15\text{-}106$	#asgs	1.9K	1.7K	21.2K	23.3K
$(100/100$ sat)	#slvs	100	100	100	100
$bqwh$	cpu	20.12	8.9	705.07	719.16
$18\text{-}141$	#asgs	25.8K	17.7K	$1,990.8\text{K}$	$2,143.3\text{K}$
$(100/100$ sat)	#slvs	100	100	98	98
$<80, 10,$	cpu	315.37	149.73	153.85	516.43
$400, 0.35>$	#asgs	373.4K	178.4K	196.4K	715.6K
$(40/100$ sat)	#slvs	100	100	100	100
$<200, 10,$	cpu	79.45	52.39	96.72	$2,317.23$
$500, 0.55>$	#asgs	62.7K	44K	94.5K	$2,810.1\text{K}$
$(14/100$ sat)	#slvs	100	100	100	51
$<900, 10,$	cpu	16.64	7.79	41.27	$2,073.24$
$1250, 0.70>$	#asgs	3.9K	2.8K	17.9K	$1,107.3\text{K}$
$(43/100$ sat)	#slvs	100	100	100	44

TAB. 5 – Instances aléatoires pures et structurées

5 Travaux connexes

La notion de pondération dynamique a été introduite en premier lieu par [Mor93] et [SK93] pour améliorer la performance des algorithmes de recherche locale. La méthode *breakout* de [Mor93] consiste simplement à augmenter le poids des tuples interdits (*no-goods*) courants lorsque l'algorithme de recherche aboutit à un minimum local. Cette pondération est alors utilisée pour s'extraire des minima locaux. De manière indépendante, [SK93] ont proposé d'incrémenter le poids des clauses (d'une formule booléenne sous forme normale conjonctive) non satisfaites par une assignation. Cette stratégie de pondération (combinée à deux autres stratégies) a permis d'accroître considérablement la performance d'algorithmes de recherche locale de type GSAT. Enfin, toujours dans le contexte de l'utilisation d'une recherche locale, [VT96] proposent d'introduire des pénalités comme termes de la fonction d'objectif d'un problème d'optimisation sous contraintes et [Tho00] utilise le principe de pondération des contraintes pour la résolution des CSPs.

D'autre part, [MSG98] proposent une technique de recherche hybride qui combine un algorithme de type GSAT et la procédure bien connue de Davis et Putnam (DP). La stratégie de branchement de l'algorithme complet DP repose sur sur une technique de pondération des contraintes gérée par GSAT dans le but d'orienter la recherche vers le noyau inconsistant du problème traité. [BGS99] ont amélioré les heuristiques de branchement pour SAT en utilisant les clauses qui se sont avérées insatisfiables auparavant. [BS00] utilisent également la pondération des clauses pour détecter les sous formules insatisfiables dans les instances SAT.

Enfin, parmi les heuristiques spécialisées proposées dans la littérature, et proches de nos travaux, on peut citer les heuristiques *LOFF* [SSX95] and *ORR* [SF96]. Ces heuristiques, spécifiques aux problèmes de *job-shop*, donnent la priorité aux variables qui utilisent les ressources critiques, et qui ont donc le plus de chances d'apparaître dans un conflit.

6 Conclusion

Dans ce papier, nous avons introduit une nouvelle heuristique générique de choix de variables qui permet d'orienter le choix de la variable à assigner en fonction des échecs rencontrés au cours de la recherche. En nous inspirant des travaux de [Mor93, SK93, MSG98, BGS99, Tho00, BS00], nous avions imaginé qu'une telle heuristique permettrait de focaliser la recherche sur les parties difficiles ou inconsistantes des CSPs. Nous avons montré, par des expérimentations sur une large classe de problèmes réels, académiques et aléatoires, que cette approche est dans les faits la plus efficace (même si, bien sûr, ponctuellement elle peut être surclassée par certaines heuristiques classiques). Ceci étant, différentes pistes peuvent être envisagées. En premier lieu, il est important de comparer cette approche aux techniques sophistiquées de retour-arrière [DF02, CvB01, JDB00, Bac00] qui permettent également d'éviter le thrashing (voir, à ce propos [LBH04b]). En second lieu, l'étude présentée ici, permet d'envisager d'autres heuristiques plus fines, utilisant conjointement les échecs et les succès pour décider tant du choix des variables que de celui des valeurs.

Remerciements

Nous tenons à remercier Carla Gomes pour avoir mis à notre disposition les outils nécessaires à la génération des instances bQWH. Ce travail a été soutenu, en partie, par l'IUT de Lens, le CNRS et la région Nord-Pas de calais dans le cadre du programme TACT.

Références

[Bac00] F. Bacchus. Extending forward checking. In *Proceedings of CP'00*, pages 35–51, 2000.

[BCS01] C. Bessiere, A. Chmeiss, and L. Sais. Neighborhood-based variable ordering heuristics for the constraint satisfaction problem. In *Proceedings of CP'01*, pages 565–569, 2001.

[Bes94] C. Bessiere. Arc consistency and arc consistency again. *Artificial Intelligence*, 65 :179–190, 1994.

[BFR99] C. Bessiere, E.C. Freuder, and J. Regin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107 :125–148, 1999.

[BGS99] L. Brisoux, E. Gregoire, and L. Sais. Improving backtrack search for sat by means of redundancy. In *Proceedings sof the 11th International Symposium on Methodologies for Intelligent Systems (ISMIS'99)*, pages 301–309, 1999.

[BLW89] M. Beresin, E. Levin, and J. Winn. A chessboard coloring problem. *The College Mathematics Journal*, 20(2) :106–114, 1989.

[BR96] C. Bessiere and J. Regin. MAC and combined heuristics : two reasons to forsake FC (and CBJ ?) on hard problems. In *Proceedings of CP'96*, pages 61–75, 1996.

[BR01] C. Bessiere and J. Regin. Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI'01*, pages 309–315, 2001.

[Bre79] D. Brelaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22 :251–256, 1979.

[BS00] R. Bruni and A. Sassano. Detecting minimaly unsatisfiable subformulae in unsatisfiable SAT instances by means of adaptative core search. In *Proceedings of SAT'00*, 2000.

[CvB01] X. Chen and P. van Beek. Conflict-directed backjumping revisited. *Journal of Artificial Intelligence Research*, 14 :53–81, 2001.

[DF02] R. Dechter and D. Frost. Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136 :147–188, 2002.

[DM89] R. Dechter and I. Meiri. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In *Proceedings of IJCAI'89*, pages 271–277, 1989.

[FD95] D. Frost and R. Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of IJCAI'95*, pages 572–578, 1995.

[Fre82] E.C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1) :24–32, 1982.

[GMP+96] I.P. Gent, E. MacIntyre, P. Prosser, B.M. Smith, and T. Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Proceedings of CP'96*, pages 179–193, 1996.

[GS02] C.P. Gomez and D. Shmoys. Completing quasigroups or latin squares : a structured graph coloring problem. In *Proceedings of Computational Symposium on Graph Coloring and Generalization*, 2002.

[GSCK00] C.P. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24 :67–100, 2000.

[HE80] R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14 :263–313, 1980.

[HH00] M.C. Horsch and W.S. Havens. An empirical study of probabilistic arc consistency as a variable ordering heuristic. In *Proceedings of CP'00*, pages 525–530, 2000.

[JDB00] N. Jussien, R. Debruyne, and P. Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Proceedings of CP'00*, pages 249–261, 2000.

[LBH03a] C. Lecoutre, F. Boussemart, and F. Hemery. Au coeur de la consistance d'arc. In *Actes des 9èmes Journées Nationales sur la Résolution Pratique des Problèmes NP-Complets (JNPC'03)*, pages 233–247, Amiens, France, June 2003.

[LBH03b] C. Lecoutre, F. Boussemart, and F. Hemery. Exploiting multidirectionality in coarse-grained arc consistency algorithms. In *Proceedings of CP'03*, pages 480–494, 2003.

[LBH04a] C. Lecoutre, F. Boussemart, and F. Hemery. Abscon 2.1, a constraint programming platform. http ://www.cril.univ-artois.fr/˜lecoutre, November 2004.

[LBH04b] C. Lecoutre, F. Boussemart, and F. Hemery. Techniques de retour-arrière intelligent versus heuristiques dirigées par les conflits. In *Actes de JNPC'04*, 2004.

[Mac77] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8 :118–126, 1977.

[MH86] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28 :225–233, 1986.

[Mor93] P. Morris. The breakout method for escaping from local minima. In *Proceedings of AAAI'93*, pages 40–45, 1993.

[MSG98] B. Mazure, L. Sais, and E. Gregoire. Boosting complete techniques thanks to local search methods. *Annals of Mathematics and Artificial Intelligence*, 22 :319–331, 1998.

[Nad88] B.A. Nadel. Tree search and arc consistency in constraint satisfaction algorithms. *Search in Artificial Intelligence*, pages 287–342, 1988.

[SF94] D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the PPCPA'94*, Seattle WA, 1994.

[SF96] N. Sadeh and M.S. Fox. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence*, 86 :1–41, 1996.

[SG98] B.M. Smith and S.A. Grant. Trying harder to fail first. In *Proceedings of ECAI'98*, pages 249–253, Brighton, UK, 1998.

[SK93] B. Selman and H. Kautz. Domain-independent extensions to gsat : solving large structured satisfiability problems. In *Proceedings of IJCAI'93*, pages 290–295, 1993.

[Smi99] B.M. Smith. The brelaz heuristic and optimal static orderings. In *Proceedings of CP'99*, pages 405–418, Alexandria, VA, 1999.

[SSX95] N. Sadeh, K. Sycara, and Y. Xiong. Backtracking techniques for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence*, 76 :455–480, 1995.

[Tho00] J.R. Thornton. *Constraint weighting local search for constraint satisfaction*. PhD thesis, Griffith University, Australia, 2000.

[vD02] M.R.C. van Dongen. $AC3_d$ an efficient arc consistency algorithm with a low space complexity. In *Proceedings of CP'02*, pages 755–760, 2002.

[VT96] C. Voudouris and E. Tsang. Partial constraint satisfaction problems and guided local search. In *Proceedings of second international conference on practical application of constraint technology (PACT'96)*, pages 337–356, 1996.

[WGS03] R. Williams, C.P. Gomes, and B. Selman. Backdoors to typical case complexity. In *Proceedings of IJCAI'03*, 2003.

[ZY01] Y. Zhang and R.H.C. Yap. Making AC3 an optimal algorithm. In *Proceedings of IJCAI'01*, pages 316–321, Seattle WA, 2001.