

Configuration automatique d'un solveur générique intégrant des techniques de décomposition arborescente pour la résolution de problèmes de satisfaction de contraintes

Loïc Blet

▶ **To cite this version:**

Loïc Blet. Configuration automatique d'un solveur générique intégrant des techniques de décomposition arborescente pour la résolution de problèmes de satisfaction de contraintes. Intelligence artificielle [cs.AI]. INSA de Lyon, 2015. Français. NNT : 2015ISAL0085 . tel-01214086v2

HAL Id: tel-01214086

https://theses.hal.science/tel-01214086v2

Submitted on 24 Feb 2016

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

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

INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE LYON

N° attribué par la bibliothèque
2015ISAL0085







THÈSE

pour obtenir le grade de
DOCTEUR de INSA Lyon
Spécialité : **Informatique**
préparée au **Laboratoire d'InfoRmatique en Images et Système d'information**
dans le cadre de l'École Doctorale **Math-Info**

présentée et soutenue publiquement
par

Loïc Blet

le 30 septembre 2015

Titre:
Configuration automatique d'un solveur générique intégrant des techniques de décomposition arborescente pour la résolution de problèmes de satisfaction de contraintes.

Directrice de thèse: **Christine Solnon, Professeure, INSA Lyon, LIRIS**
Co-directeur de thèse: **Samba Ndojh Ndiaye, Maître de conférences, Univ. Lyon 1, LIRIS**

Jury

M. Philippe Jégou, Professeur, Université Aix Marseille, LSIS,	Président du jury
Mme. Christine Solnon, Professeure, INSA Lyon, LIRIS,	Directrice de thèse
M. Samba Ndojh Ndiaye, Maître de conférences, Université Lyon 1, LIRIS,	Co-Directeur de thèse
M. Yves Deville, Professeur, Université Catholique de Louvain, BeCool,	Rapporteur
M. Pierre Flener, Professeur, Université d'Uppsala, ASTRA,	Rapporteur

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Remerciements

En tout premier, je tiens à remercier mes encadrants, Christine et Samba, qui ont été des piliers incroyables pour moi. Soutenu à la fois par la précision de Samba et la vision de Christine ; par le calme olympien de Samba et les encouragements lumineux et limpides de Christine. Merci.

Merci aux rapporteurs de cette thèse, Yves Deville et Pierre Flener. J'ai énormément apprécié les moments passés avec les équipes BeCool et ASTRA dont ils font parti. En plus, Pierre – excellent professeur– m'a fait découvrir le domaine de la programmation par contraintes, lors de mon année Érasmus en Suède, qui m'a amené, après quelques péripéties, à entreprendre cette thèse.

Pour la vie au travail, les pauses cafés et les nombreuses fois où nous avons refait le monde, milles merci à mes collègues et mes cobureaux, qui ont du me supporter jour après jour. J'en oublierai forcément (pardon) mais voici un échantillon représentatif que je remercie : Stéphane, Camille, Elsa, Jean-David, Jérémy, François, Nicolas, Nadia, Azhar, Sébastien, Manel, Vincent, Vincent (l'autre), Diana, Mazen, Thomas, Amadou, Rémi, John et David.

En particulier, merci à mes derniers cobureaux, qui m'ont subi pendant la phase de rédaction du manuscrit : Romain, Julien, Maël et Vanessa. Une vraie chance de partager un bureau avec vous !

Je profite de cette page pour exprimer ma gratitude envers ma famille, chose évidente que je fais bien trop rarement. Merci donc à ma mère, qui m'a appris à être pragmatique et à relativiser les événements ; à mon père qui me transmet son esprit combatif et sa culture. Merci Alice, c'est rassurant de te parler régulièrement (et de parler de nos thèses !). Merci Cyril, en tant que petit frère, j'ai toujours l'impression de devoir te rattraper et te dépasser.

À mes amis de longue date, vous m'enrichissez plus que vous ne l'imaginez. Théo, Thomas, Louise ça a été un bonheur de vous revoir à chaque retour à Paris. Marion et Gaël, j'adore toujours autant votre compagnie (ne vous liguez pas trop contre moi en jouant à Carcassone quand même).

Un énorme merci à tout les membres du club de Go de Lyon, qui m'ont permis de griller ce qu'il me reste de neurones après la thèse en jouant à ce merveilleux jeu ! Merci à ceux avec qui j'ai partagé le plus de moments : Fritz, Rémi, Frédéric P., Caroline, Jonathan, J-C, Florent R., Florent L. et Léo.

Et pour le groupe de bons vivants avec qui j'apprends la passion du jeu de théâtre, merci, merci, merci. En particulier, merci à Didier, le professeur qui m'apprends à écouter et à ressentir ; et à mes partenaires de scène ces deux dernières années : Nathalie, Ingrid et Delphine.

Enfin, un merci tout particulier à Hélène, qui a su être là pour moi pendant les derniers mois remuants de cette thèse.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Table des matières

1 Introduction **3**
- 1.1 Contexte . . . 3
- 1.2 Contributions . . . 4
- 1.3 Plan . . . 6

I Contexte **9**

2 Programmation par contraintes **11**
- 2.1 Problèmes de satisfaction de contraintes . . . 11
- 2.2 Algorithmes de résolution de CSP . . . 14
  - 2.2.1 Arbres de recherche . . . 15
  - 2.2.2 Propagation des contraintes et cohérences locales . . . 15
  - 2.2.3 Stratégies de retour arrière . . . 17
  - 2.2.4 Exploiter statiquement la structure du problème . . . 27
  - 2.2.5 Heuristiques de choix de variable . . . 30
- 2.3 Discussion . . . 32

3 Programmation par optimisation **33**
- 3.1 Apprentissage artificiel . . . 34
  - 3.1.1 Représentation des objets de l'apprentissage . . . 34
  - 3.1.2 Classification supervisée . . . 34
  - 3.1.3 Régression . . . 36
  - 3.1.4 Clustering . . . 37
- 3.2 L'apprentissage pour la programmation par contraintes . . . 38
  - 3.2.1 Configuration . . . 39
  - 3.2.2 Portfolios . . . 40
  - 3.2.3 Sélection . . . 40
  - 3.2.4 Ordonnancement . . . 41
  - 3.2.5 Approches modifiant dynamiquement la stratégie de recherche . . . 43
  - 3.2.6 Apprentissage par renforcement pour les heuristiques . . . 43
- 3.3 Discussion . . . 43

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

II Un cadre générique pour les CSP **45**

4 Solveur générique **47**
4.1 Présentation du solveur générique . . . 47
4.2 Rappel de l'algorithme de [LBH04] . . . 48
4.3 Extensions de l'algorithme à de nouvelles stratégies dynamiques . . . 53
4.3.1 Inclure Decision Repair . . . 54
4.3.2 Inclure Conflict directed BackJumping with Reordering . . . 55
4.4 Intégration de BTD . . . 56
4.5 Implémentation . . . 57
4.5.1 Pré-traitement sur les instances . . . 57
4.5.2 Non déterminisme . . . 57
4.6 Discussion . . . 58

5 Instances considérées pour l'évaluation **61**
5.1 Instances au format XCSP . . . 61
5.1.1 Élimination de certaines instances . . . 62
5.1.2 Les différentes classes d'instances . . . 62
5.2 Instances structurées générées aléatoirement . . . 64
5.2.1 Méthode de génération . . . 64
5.2.2 Paramètres retenus pour générer un ensemble d'instances . . . 66
5.3 Discussion . . . 66

6 Résultats expérimentaux **69**
6.1 Évaluation de l'exploitation de décompositions arborescentes avec des approches de retours arrières dynamique . . . 69
6.2 Comparaison avancée de vingt-quatre configurations . . . 72
6.2.1 Protocole expérimental pour la comparaison de configurations . . . 72
6.2.2 Comparaison des taux de succès . . . 74
6.2.3 Comparaison du nombre d'instances « bien résolues » . . . 75
6.2.4 Étude de la répartition en classes des instances bien résolues . . . 77
6.2.5 Décomposer ou ne pas décomposer ? . . . 78
6.3 Discussion . . . 79

III Sélection automatique de configuration **81**

7 Sélection de solveur dans un portfolio **83**
7.1 Cadre basique du sélecteur . . . 83
7.1.1 Propriétés utilisées pour décrire une instance CSP . . . 84
7.1.2 Entraînement . . . 85
7.2 Choix d'un sous-ensemble de configurations . . . 85
7.3 Discussion . . . 89

8 Évaluation expérimentale **91**
8.1 Protocole expérimental . . . 91
8.2 Qualité des configurations sélectionnées . . . 92
8.3 Comparaison des taux de succès . . . 93
8.4 Discussion . . . 95

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

9 Conclusion **97**
9.1 Bilan 97
9.2 Perspectives 98

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Résumé

La programmation par contraintes intègre des algorithmes de résolution génériques dans des langages de modélisation déclaratifs basés sur les contraintes : ces langages permettent de décrire des problèmes combinatoires sous la forme d'un ensemble de variables devant prendre leurs valeurs dans des domaines en satisfaisant des contraintes.

De nombreux problèmes réels peuvent être modélisés de cette façon comme, par exemple, les problèmes de planification, d'ordonnancement, de découpe, etc. Ces problèmes sont NP-complets dans le cas général de domaines finis.

Nous introduisons dans cette thèse un algorithme de résolution générique qui est paramétré par :

— une stratégie d'exploration de l'espace de recherche, à choisir parmi les six stratégies suivantes, *chronological backtracking*, *conflict directed backjumping*, *conflict directed backjumping with reordering*, *dynamic backtracking*, *decision repair*, et *backtracking with tree decomposition* ;
— une heuristique de choix de variables, à choisir parmi deux heuristiques, à savoir, *min-domain/ddeg* et *min-domain/wdeg* ;
— une technique de propagation de contraintes, à choisir parmi deux techniques, à savoir, *forward checking* et *maintaining arc consistency*.

Ainsi, cet algorithme générique s'instancie en vingt-quatre configurations différentes ; certaines correspondant à des algorithmes connus, d'autres étant nouvelles. Ces vingt-quatre configurations ont été comparées expérimentalement sur un benchmark de plus de mille instances, chaque configuration étant exécutée plusieurs fois sur chaque instance pour tenir compte du non déterminisme des exécutions. Des tests statistiques sont utilisés pour comparer les performances. Cette évaluation expérimentale a permis de mieux comprendre la complémentarité des différents mécanismes de résolution, avec une attention particulière portée sur la capacité à tirer parti de la structure des instances pour accélérer la résolution. Nous identifions notamment treize configurations complémentaires telles que chaque instance de notre benchmark est bien résolue par au moins une des treize configurations.

Une deuxième contribution de la thèse est d'introduire un sélecteur capable de choisir automatiquement la meilleure configuration de notre algorithme générique pour chaque nouvelle instance à résoudre : nous décrivons chaque instance par un ensemble de descripteurs et nous utilisons des techniques d'apprentissage automatique pour construire un modèle de choix de configuration à partir de ces descripteurs. Sachant que l'apprentissage est généralement plus difficile quand il y a beaucoup de configurations, nous exprimons le problème du choix du sous-ensemble de configurations pouvant être sélectionnées comme un problème de couverture d'ensemble et nous comparons deux critères de choix : le premier vise à maximiser le nombre d'instances résolues par au moins une configuration et le second vise à maximiser le nombre d'instances pour lesquelles il y a une bonne configuration disponible. Nous montrons expérimentalement que la seconde stratégie obtient généralement de meilleurs résultats, et que le sélecteur obtient de meilleures performances que chacune de nos vingt-quatre configurations initiales.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Abstract

Constraint programming integrates generic solving algorithms within declarative languages based on constraints : these languages allow us to describe combinatorial problems as a set of variables which have to take their values in domains while satisfying constraints.

Numerous real-life problems can be modelled in such a way, such as for instance, planification problems, scheduling problems, ... These problems are NP-complete in the general case of finite domains.

We introduce in this work a generic solving algorithm parameterized by :

— a strategy for exploring the search space, to be chosen from the following six, chronological backtracking, conflict directed backjumping, conflict directed backjumping with reordering, dynamic backtracking, decision repair, and backtracking with tree decomposition ;

— a variable ordering heuristic, to be chosen from the following two, min-domain/ddeg and min-domain/wdeg ;

— a constraint propagation technique, to be chosen from the following two, forward checking and maintaining arc consistency.

Thus, this algorithm leads to 24 different configurations ; some corresponding to already known algorithms, others being new. These 24 configurations have been compared experimentally on a benchmark of more than a thousand instances, each configuration being executed several times to take into account the non-determinism of the executions, and a statistical test has been used to compare the performance. This experimental evaluation allowed us to better understand the complementarity of the different solving mechanisms, with a focus on the ability to exploit the structure of the instances to speed up the solving process. We identify 13 complementary configurations such that every instance of our benchmark is well solved by at least one of the 13 configurations.

A second contribution of this work is to introduce a selector able to choose automatically the best configuration of our generic solver for each new instance to be solved : we describe each instance by a set of features and we use machine learning techniques to build a model to choose a configuration based on these features. Knowing that the learning process is generally harder when there are many configurations to choose from, we state the problem of choosing a subset of configurations that can be picked as a set covering problem and we compare two criteria : the first one aims to maximize the number of instances solved by at least one configuration and the second one aims to maximize the number of instances for which there is a good configuration available. We show experimentally that the second strategy obtains generally better results and that the selector obtains better performances than each of the 24 initial configurations.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Chapitre 1

Introduction

Sommaire

1.1 Contexte 3
1.2 Contributions 4
1.3 Plan 6

1.1 Contexte

Nous rencontrons tous les jours des problèmes combinatoires, pouvant être résolus en énumérant un nombre fini de combinaisons. L'approvisionnement des vélos à louer en ville, le calcul de trajets multi-modaux, la conception d'emplois du temps sont des problèmes combinatoires dont la résolution « à la main » est une tâche dantesque pour des problèmes d'une échelle pourtant encore raisonnable. De fait, ces problèmes sont bien souvent NP-difficiles ce qui implique que leur résolution se heurte à un phénomène d'explosion combinatoire du nombre de combinaisons à explorer.

Il existe plusieurs façons de modéliser ces problèmes. Nous pouvons citer la programmation linéaire qui propose de traduire les contraintes du problème sous forme d'équations et d'inéquations linéaires. Les problèmes de satisfaction de formules booléennes (SAT) sont une modélisation avec des variables logiques booléennes, liées dans des clauses logiques avec les opérateurs ET, OU et NON.

Les problèmes de satisfaction de contraintes sont une autre modélisation plus expressive. Ils décrivent le problème en listant ses variables ainsi que les valeurs possibles pour ces variables. Les contraintes sont ensuite posées pour restreindre les valeurs pouvant être simultanément affectées à des sous-ensembles de variables. Une solution du problème de satisfaction de contraintes est une affectation, d'une valeur à chaque variable, respectant toutes les contraintes du problème.

Ce type de modélisation offre une expressivité intéressante, grâce aux contraintes liant les variables du problème. En effet nous disposons, par exemple, pour ces modèles de contraintes :

— booléennes comme $x \wedge y = VRAI$,
— arithmétiques comme $x + y < 5$.

La programmation par contraintes est un domaine proposant des langages déclaratifs permettant de décrire des problèmes de satisfaction de contraintes et intégrant des méthodes pour résoudre ces problèmes. De nombreuses méthodes de résolution ont

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

été proposées et améliorées au fil des recherches. L'algorithme fondamental de résolution est le retour-arrière chronologique. Cet algorithme propose d'explorer l'espace des combinaisons candidates, appelé espace de recherche, en construisant un arbre de recherche dont les nœuds internes correspondent à des choix d'affectations de valeurs à des variables et les feuilles à des affectations partielles ne pouvant être étendues sans violer de contraintes (donc des échecs), ou à des affectations totales satisfaisant toutes les contraintes (donc des solutions). Le retour-arrière chronologique consiste à explorer cet arbre de recherche en profondeur d'abord jusqu'à trouver une solution, ou avoir terminé l'exploration sans avoir trouvé de solution (et prouvé l'incohérence de l'instance).

Après cet algorithme de retour-arrière chronologique (*Chronological Backtracking* ou CBT), des améliorations visant à éviter les redondances dans la recherche ont été proposées en exploitant dynamiquement la structure des instances. Par exemple *Conflict directed BackJumping* (CBJ) enregistre certaines informations durant la recherche pour revenir plus rapidement, lors d'un échec, à un nœud plus haut dans l'arbre de recherche. Pour donner plus d'importance à l'heuristique de choix de variable, le *Conflict directed BackJumping with variable Re-ordering* (CBJR) peut – sous certaines conditions – remonter une variable dans l'affectation courante. Le *Dynamic BackTracking* (DBT) poursuit cette idée en essayant en plus de conserver certaines informations, lors d'un échec, plutôt que de détruire une partie du travail effectué (l'espace de recherche n'est plus structuré en arbre pour DBT mais en graphe). L'algorithme *Decision Repair* (DR) est une variante offrant plus de souplesse dans la recherche en autorisant à remettre en cause n'importe quelle décision.

D'un autre côté des méthodes visant à exploiter statiquement la structure des problèmes ont été proposées. Comme il a été remarqué que certains problèmes industriels ont une structure intéressante, des méthodes de décomposition issues de la théorie des graphes sont utilisées pour capturer la structure de ces problèmes. Une méthode comme le retour-arrière avec décomposition arborescente (*Backtracking with Tree Decomposition* ou BTD) peut ensuite être appliquée pour exploiter cette structure.

Par ailleurs, différentes heuristiques peuvent être considérées pour choisir, à chaque nœud de l'arbre, la prochaine décision de branchement à effectuer. Enfin, la taille de l'arbre de recherche peut être réduite en propageant les contraintes à chaque nœud afin de vérifier la cohérence du sous-problème associé au nœud. Différents algorithmes de propagation correspondant à différents niveaux de cohérence locale, peuvent être utilisés, les méthodes les plus connues étant *Forward Checking* (FC) et *Maintaining Arc Consistency* (MAC).

1.2 Contributions

Bien que nombre de ces méthodes aient été comparées entre elles, une étude de grande envergure les rassemblant toutes n'a jamais été réalisée. Nous proposons donc de rassembler un ensemble varié d'instances de problèmes de satisfaction de contraintes pour étudier le comportement et l'efficacité de ces méthodes. Nous pourrons ainsi confirmer les précédentes études ainsi qu'apporter de nouveaux éléments de distinction entre ces méthodes.

Il est intéressant, pour réaliser cette comparaison, de poser un cadre générique et uniforme pour ces algorithmes. Nous étendrons pour ce faire un cadre existant en y ajoutant *Decision Repair*, *Conflict directed BackJumping with Re-ordering* et l'exploitation de la décomposition arborescente. L'extension du cadre va nous permettre de

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

proposer et d'évaluer de nouveaux algorithmes pour résoudre les CSP. Ce sont des combinaisons de méthodes de retour arrière, d'heuristiques de choix de variables, de techniques de propagation de contraintes et d'exploitation de la structure statique. En particulier, les retours arrières CBJ et DR n'avaient pas été testés avec BTD. Plus minime, CBJR n'avait pas été testé avec la propagation MAC.

Ce cadre générique nous permet de comparer différents algorithmes de résolution sur une même base d'implémentation. Nous sommes intéressés en particulier par une étude des différences et complémentarités dans l'exploitation dynamique ou statique de la structure, ou une combinaison des deux.

En tout nous disposons de vingt-quatre configurations. Nous les avons évaluées sur un benchmark de plus de mille instances ; chaque configuration étant exécutée plusieurs fois sur chaque instance pour tenir compte de l'aspect non-déterministe des exécutions. Souligner la nature non-déterministe de ces configurations est une petite contribution car trop souvent des choix arbitraires sont faits pour rendre les algorithmes déterministes. De plus, disposer de plusieurs exécutions nous permet d'utiliser des tests statistiques pour comparer les performances des différentes configurations. Plus précisément, nous introduisons la notion d'instance « bien résolue » par une configuration, lorsque les résultats obtenus par cette configuration ne sont pas significativement différents de ceux obtenus avec la meilleure configuration, selon un test de Student.

Grâce à cette évaluation expérimentale nous analysons la complémentarité des différents mécanismes composant un algorithme de résolution de CSP. Nous détaillons notamment la capacité à tirer parti de la structure des instances pour accélérer la résolution. Nous faisons une distinction entre l'exploitation statique de la structure, comme le fait BTD, de l'exploitation dynamique de la structure comme le font CBJ, CBJR, DBT et DR pour les mécanismes de retour-arrière, et *ddeg* mais surtout *wdeg* pour les heuristiques de choix de variables.

Nous remarquons que treize configurations – sur les vingt quatre de départ – suffisent pour que chaque instance de notre benchmark soit bien résolue par au moins une des treize configurations complémentaires.

La question de choisir la meilleure méthode parmi toutes celles étudiées se pose naturellement. Pour sélectionner efficacement une méthode pour une instance, nous nous tournons vers l'apprentissage artificiel et notamment les algorithmes de classification pour la sélection automatique de configuration pour une instance donnée. Nous décrivons chaque instance de notre benchmark par un ensemble de descripteurs et nous utilisons un classifieur pour construire un modèle de choix de configuration à partir de ces descripteurs.

Comme l'apprentissage est généralement plus difficile quand il y a beaucoup de configurations parmi lesquelles choisir, nous nous sommes posés la question de choisir un sous-ensemble de configurations pertinentes afin de faciliter le processus de classification. Nous proposons une méthode de couverture d'ensemble s'appuyant sur les résultats de notre comparaison expérimentale des différentes méthodes de résolution de problèmes de satisfaction de contraintes. Nous comparons deux critères de choix pour la couverture d'ensemble : le premier vise à maximiser le nombre d'instances résolues par au moins une configuration (avant un temps limite fixé) et le second vise à maximiser le nombre d'instances pour lesquelles il y a une bonne configuration disponible. Nous montrons expérimentalement que la seconde stratégie choisit plus souvent de meilleures configurations pour chaque instance et également que cette seconde stratégie permet de résoudre plus d'instances de notre benchmark que la première stratégie et que chacune de nos vingt-quatre configurations initiales.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

1.3 Plan

Nous commencerons dans le chapitre 2 à poser le vocabulaire des problèmes de satisfaction de contraintes et de la programmation par contraintes. Nous détaillerons ici les différentes composantes d'un algorithme pour résoudre des problèmes de satisfaction de contraintes. Nous expliquerons ainsi les méthodes de propagation de contraintes visant à se servir des contraintes pour filtrer certaines valeurs des variables. Nous verrons deux heuristiques pour choisir la prochaine variable à affecter lors de la recherche. Ensuite nous y présenterons les algorithmes mentionnés ci-avant, qui ont chacun des stratégies différentes pour gérer les échecs lors de la recherche. Nous introduirons enfin la décomposition arborescente d'un CSP et son exploitation via l'algorithme *Backtracking with Tree-Decomposition* (BTD).

Le chapitre 3 sera l'occasion pour nous de présenter l'autre pan de nos travaux, à savoir les outils que nous utiliserons venant de l'apprentissage artificiel. Nous décrirons ainsi les méthodes de classification que nous utiliserons ensuite et ferons une revue des travaux utilisant l'apprentissage pour la programmation par contraintes. Il sera question d'algorithmes de sélection, d'ordonnancement ou encore de configuration de solveurs. Nous mentionnerons aussi les possibilités de modifier dynamiquement la stratégie de recherche.

Viendra ensuite le chapitre 4 où nous exposerons notre cadre générique s'instanciant en trente-deux configurations au total. Notamment nous proposerons notre technique pour lier l'exploitation de la décomposition arborescente aux méthodes de retour-arrière intelligent. Nous préciserons les conditions d'implémentation de ce cadre.

Le chapitre 5 décrit l'ensemble de plus de mille instances considérées pour notre étude expérimentale, en présentant une méthode de génération aléatoire de problèmes ainsi qu'une base de données d'instances dédiée précédemment à des compétitions académiques.

Nous présenterons au chapitre 6 les résultats de l'étude expérimentale des différentes configurations sur l'ensemble d'instances qui aura été présenté au chapitre précédent. Dans une première expérimentation, nous évaluerons l'intérêt de coupler des mécanismes de retour-arrière intelligents avec une exploitation de décomposition arborescente, et nous montrerons que ces combinaisons n'améliorent généralement pas la résolution. Dans une deuxième expérimentation, nous comparerons de façon intensive vingt-quatre configurations de notre cadre générique. Chaque couple de configuration et d'instances sera testé quinze fois pour prendre en compte l'aspect non déterministe des configurations. Grâce à ces résultats nous pourrons comparer les configurations et donner une notion de configuration dominée. Nous nous demanderons s'il existe un critère simple permettant de décider si l'exploitation d'une décomposition arborescente pour une instance donnée sera bénéfique.

Ayant les résultats des configurations disponibles, le chapitre 7 présentera le cadre d'un sélecteur de configuration qui pourra en tirer parti. Un point sensible d'un sélecteur est l'ensemble de configurations parmi lequel le sélecteur doit choisir. Nous proposerons une nouvelle méthode pour choisir cet ensemble de configurations en nous basant sur des tests statistiques rendus possibles par l'étude expérimentale.

Enfin, le chapitre 8 sera l'occasion de détailler les performances des sélecteurs que nous avons introduits. Nous commencerons par donner le protocole de comparaison avant d'étudier les résultats obtenus. Les sélecteurs seront évalués à la fois sur la qualité des configurations sélectionnées ainsi que sur leurs taux de succès pour la résolution des problèmes.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Nous pourrons finalement récapituler nos avancées dans le chapitre 9 et envisager les perspectives de nos travaux.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Première partie

Contexte

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Chapitre 2

Programmation par contraintes

Sommaire

2.1 Problèmes de satisfaction de contraintes 11
2.2 Algorithmes de résolution de CSP 14
2.2.1 Arbres de recherche 15
2.2.2 Propagation des contraintes et cohérences locales 15
2.2.3 Stratégies de retour arrière 17
2.2.4 Exploiter statiquement la structure du problème 27
2.2.5 Heuristiques de choix de variable 30
2.3 Discussion 32

Les contraintes font partie de notre vie quotidienne, qu'il s'agisse par exemple de faire un emploi du temps, de remplir un camion de déménagement avec des cartons de tailles diverses, ou encore de planifier un trafic aérien.

Dans ce chapitre, nous allons détailler les mécanismes de la programmation par contraintes qui permettent de modéliser et résoudre des problèmes définis en terme de contraintes. Nous débuterons la présentation par le formalisme des problèmes de satisfaction de contraintes en section 2.1. Ensuite nous listerons les éléments constitutifs d'un algorithme de résolution de problèmes de satisfaction de contraintes performant en section 2.2, à savoir la propagation des contraintes, les heuristiques de choix de variable, la méthode d'exploration de l'espace de recherche et enfin l'exploitation de la structure du problème à résoudre.

La majorité des définitions et des exemples de ce chapitre proviennent du livre [Lec09].

2.1 Problèmes de satisfaction de contraintes

Un problème de satisfaction de contraintes (*Constraint Satisfaction Problem* ou CSP) est défini par un ensemble fini de variables, chaque variable devant être affectée à une valeur de son domaine tout en respectant un ensemble de contraintes, qui restreignent les valeurs que peuvent prendre simultanément les variables.

Définition 1 (CSP)
Un problème de satisfaction de contraintes est défini par un triplet (X, D, C) *tel que :*

— X *est un ensemble fini de variables ;*

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

— *D est une fonction qui associe à chaque variable $x_i \in X$ son domaine $D(x_i)$, c'est-à-dire l'ensemble des valeurs que peut prendre x_i ;*
— *C est un ensemble de contraintes. Chaque contrainte $c_j \in C$ est une relation $rel(c_j)$ entre certaines variables de X, restreignant les valeurs que peuvent prendre simultanément ces variables.*

Nous donnons quelques définitions et notations supplémentaires concernant les contraintes pour la suite de notre exposé.

Définition 2 (portée, support, conflits et arité d'une contrainte)
La portée *d'une contrainte c, notée $portée(c)$, est l'ensemble des variables sur lesquelles c restreint les valeurs autorisées.*

L'ensemble des tuples valides d'une contrainte c, noté $val(c)$, est l'ensemble :

$$val(c) = \prod_{x \in portée(c)} D(x)$$

Ce sont tous les tuples possibles pour une contrainte, qu'ils respectent ou non la contrainte.

L'ensemble des supports *d'une contrainte c, noté $sup(c)$, est l'ensemble :*

$$sup(c) = val(c) \cap rel(c)$$

Les supports d'une contrainte sont les tuples qui respectent la relation de la contrainte. Nous parlerons aussi de support pour une contrainte comme étant un tuple appartenant à cet ensemble des supports.

L'ensemble des conflits *d'une contrainte c, noté $con(c)$, est l'ensemble :*

$$con(c) = val(c) \setminus rel(c)$$

Les conflits sont les tuples qui ne respectent pas la relation de la contrainte.

*L'*arité *d'une contrainte c est le nombre de variables sur lesquelles porte c.*

La relation $rel(c)$ peut être définie en extension, en listant l'ensemble de ses tuples de valeurs, ou en intention, en utilisant des relations mathématiques entre des expressions arithmétiques, ensemblistes ou booléennes, par exemple.

Notez que nous ne considérons que les CSP dont le domaine est fini et dont l'ensemble de contraintes est également fini.

Une contrainte est dite *binaire* si elle porte sur deux variables. À l'inverse, une contrainte portant sur plus de deux variables est dite n-aire. Par extension un CSP ne contenant que des contraintes binaires est dit *binaire*. **Dans notre étude nous ne considérons que des CSP binaires**.

Nous ne parlerons pas ici des structures de données pour représenter les domaines, bien qu'elles soient importantes et influent sur nos algorithmes de résolution. Pour une présentation de structures de données éprouvées en programmation par contraintes, le lecteur peut se référer à [Lec09].

Exemple 1 (Le problème des n-reines)
Le problème est le suivant : placer n reines sur un échiquier de taille n par n, de telle sorte qu'aucune paire de reines ne soit en position de prise. Rappelons qu'une reine peut prendre un pièce qui se situe sur la même ligne, la même colonne et les mêmes diagonales qu'elle.

Ce problème peut facilement être modélisé sous la forme d'un CSP. Plusieurs modélisations différentes sont possibles. Nous en donnons une ci-dessous.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Remarquons qu'il ne peut y avoir qu'une reine par colonne sur l'échiquier, car deux reines sur une même colonne sont en position de prise. Nous modélisons ce problème en tenant compte de cette remarque. Nous utiliserons donc n variables entières avec comme domaine les entiers allant de 1 à n. Si la variable x_i est affectée à la valeur j, cela signifie que la i^e reine est sur la colonne i et la ligne j :

- $X = \{x_1, \ldots, x_n\}$
- $D(x_i) = \{1, \ldots, n\}, \forall i \in \{1, \ldots, n\}$
- *il y a deux types de contraintes :*
  - *pour les lignes,* $\forall (i,j) \in \{1 \ldots, n\} \times \{1, \ldots, n\}, i \neq j \Rightarrow x_i \neq x_j$
  - *pour les diagonales,* $\forall (i,j) \in \{1, \ldots, n\} \times \{1, \ldots, n\}, i \neq j \Rightarrow |i-j| \neq |x_i - x_j|$

Le tuple $(1,2)$ appartient à l'ensemble des conflits de la contrainte entre x_1 et x_2

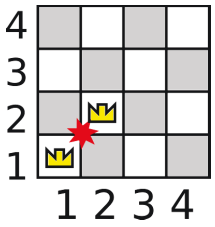


FIGURE 2.1 – *Exemple de conflit entre deux reines placées pour le problème des 4-reines*

Voir la figure 2.1 pour un exemple de conflit sur le problème des 4-reines.

Définition 3 (Graphe des contraintes)
Le graphe des contraintes d'un CSP binaire (X, D, C) est un graphe $G = (X, E)$ dont l'ensemble E d'arêtes correspond aux contraintes, c'est-à-dire :

$$E = \{(x_i, x_j) \mid \exists c \in C, portée(c) = \{x_i, x_j\}\}$$

Exemple 2 (Graphe des contraintes pour le problèmes des n-reines)
Le graphe des contraintes du problème des n-reines est le graphe complet d'ordre n car toutes les variables partagent des contraintes avec toutes les autres variables.

Pour décrire une solution d'un CSP nous avons besoin de définir une *affectation* qu'elle soit *totale* ou *partielle*, *cohérente* ou *incohérente*. Nous disons qu'une contrainte est violée quand les deux variables de sa portée sont affectées et que le couple de valeurs correspondant n'est pas dans la relation de la contrainte. Une autre façon de le voir est de considérer que cette contrainte n'a pas de support ou de la même façon que les variables de cette contrainte sont affectées à des valeurs dans son ensemble de conflits.

Définition 4 (affectation)
Une affectation *A est un ensemble de couples (variable, valeur) tel que chaque valeur est prise dans le domaine de la variable correspondante. Pour un CSP (X, D, C) une affectation sera de la forme :*
$A = \{(x_1, v_1), \ldots, (x_k, v_k)\}$ tel que pour tout $\{(x_i, v_i), (x_j, v_j)\} \subseteq A, x_i \neq x_j$, $v_i \in D(x_i)$ et $v_j \in D(x_j)$.

Une affectation est dite totale *si elle instancie toutes les variables du problème, elle est dite* partielle *si elle n'en instancie qu'une partie.*
Une affectation (totale ou partielle) est cohérente *si elle n'est en conflit avec aucune contrainte et* incohérente *si elle est en conflit avec au moins une contrainte.*

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

affectation : $\{(x_1, 2), (x_2, 4), (x_3, 1), (x_4, 3)\}$

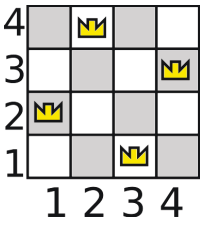


FIGURE 2.2 – *Exemple de solution pour le problème des 4-reines.*

Enfin, une *solution* est une affectation totale cohérente. Nous dirons qu'un CSP est *incohérent* s'il n'a aucune solution.

Voir la figure 2.2 pour un exemple de solution sur le problème des 4-reines.

Exemple 3 (Diverses affectations pour le problèmes des 4-reines)
Voici quelques affectations pour le problème des 4-reines :
- $\{(x_1, 1), (x_2, 1)\}$ *est partielle et incohérente,*
- $\{(x_1, 1), (x_2, 3)\}$ *est partielle et cohérente,*
- $\{(x_1, 2), (x_2, 4), (x_3, 1), (x_4, 3)\}$ *est une solution.*

Définition 5 (espace de recherche)
L'espace de recherche d'un CSP est l'ensemble de toutes ses affectations possibles, cohérentes ou incohérentes.

Notons que la taille de l'espace de recherche n'est pas nécessairement corrélée à la difficulté de la résolution du CSP. En effet, les contraintes vont servir à réduire le nombre de possibilités à explorer en enlevant certaines valeurs incohérentes du domaine de certaines variables. De plus les algorithmes que nous présentons structurent cet espace de recherche pour l'explorer le plus efficacement possible.

Résoudre un CSP est un problème NP-complet dans le cas général.

Définition 6 (good et nogood)
Un good *est une affectation partielle* $Agood$ *qui peut être étendue en une solution, c'est-à-dire qu'il existe une affectation totale cohérente* $Asol$ *telle que* $Agood \subset Asol$.
Un nogood *est une affectation partielle* $Anogood$ *qui ne peut pas être étendue en une solution, c'est-à-dire que pour toute affectation totale cohérente* $Asol$, $Anogood \nsubseteq Asol$.

Exemple 4 (Good et nogood pour le problème des 4-reines)
Voici un good et un nogood pour le problème des 4-reines :
- $\{(x_1, 1)\}$ *est un nogood, car il n'existe pas de solution avec une reine placée en (1,1) ;*
- $\{(x_1, 2), (x_2, 4)\}$ *est un good car c'est une affectation partielle pouvant être étendue en solution avec* $\{(x_3, 1), (x_4, 3)\}$.

Dans la suite nous parlerons d'une *instance* pour désigner un CSP prêt à être résolu.

2.2 Algorithmes de résolution de CSP

Il existe deux grandes approches de résolution de CSP. D'une part, les approches complètes qui explorent exhaustivement l'espace de recherche jusqu'à trouver une solution ou prouver l'absence de solution et d'autre part les approches incomplètes (ou

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

heuristiques) qui font délibérément des impasses et n'explorent qu'une partie de l'espace de recherche, comme [SKC94, LA87, GL97, MH97, DB10]. Notons que les approches incomplètes ne peuvent pas prouver l'incohérence d'une instance de façon générale. Dans ce travail, nous considérons essentiellement des approches complètes, à l'exception de l'algorithme *Decision Repair*, qui se situe à la frontière de ces deux approches.

2.2.1 Arbres de recherche

La méthode classique pour les approches complètes est la recherche arborescente qui permet d'explorer l'espace de recherche de façon systématique.

Définition 7 (arbre de recherche)
Chaque nœud interne de l'arbre correspond à une affectation partielle cohérente ; la racine est l'affectation vide ; les feuilles correspondent soit à des affectations incohérentes (des échecs) soit à des affectations totales cohérentes (des solutions).

À chaque nœud nous choisissons une variable x_i non affectée et nous créons autant de fils qu'il y a de valeurs dans le domaine $D(x_i)$ (une autre possibilité consiste à choisir une valeur $v_j \in D(x_i)$ et à créer deux fils correspondant aux choix $x_i = v$ et $x_i \neq v$, mais nous ne considérons pas ce genre d'arbre dans ce travail).

À chaque nœud de l'arbre, nous pouvons propager des contraintes pour réduire les domaines des variables non affectées et détecter des incohérences lorsque des domaines deviennent vides.

Nous pouvons construire différents arbres de recherche pour un même modèle de CSP. Ces arbres dépendent :

— de la stratégie de retour arrière considérée pour choisir le prochain nœud à développer quand nous arrivons sur un nœud échec ;
— de l'heuristique considérée pour choisir la prochaine variable à affecter ;
— et de l'algorithme utilisé pour propager les contraintes à chaque étape.

Ces trois composantes d'un algorithme de résolution de CSP sont décrites dans les trois sous-sections suivantes.

2.2.2 Propagation des contraintes et cohérences locales

Afin de réduire l'espace de recherche, nous allons présenter des techniques de filtrage basées sur des cohérences locales [Wal75, Fre78].

Les cohérences locales sont des propriétés des CSP liées à la cohérence d'un sous ensemble de variables ou de contraintes. Une cohérence locale peut être appliquée par des transformations du problème sans changer ses solutions. Ce processus s'appelle la propagation des contraintes. La propagation de contraintes s'effectue en réduisant le domaine des variables ou en créant ou renforçant des contraintes appelées *nogoods*. Cela réduit l'espace de recherche. Le problème obtenu est ainsi plus simple à résoudre. Il est possible de détecter l'incohérence d'un CSP pendant ce processus, notamment lorsque le domaine d'une variable devient vide.

Nous allons définir deux cohérences classiques : la cohérence de nœud et la cohérence d'arc.

Définition 8 (cohérence de nœud)
Une variable x est nœud-cohérente si chaque contrainte unaire c portant sur x est satisfaite par toutes les valeurs dans le domaine de la variable.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

La condition de cohérence de nœud peut être appliquée trivialement en réduisant le domaine de chaque variable aux valeurs qui satisfont toutes les contraintes unaires sur cette variable. Ainsi, les contraintes unaires peuvent être directement intégrées dans les domaines.

Exemple 5
Étant données une variable v avec un domaine $\{1,2,3,4\}$ et une contrainte $v \leq 3$, la nœud-cohérence restreint le domaine à $\{1,2,3\}$ et la contrainte sera ensuite abandonnée ; ce pré-traitement simplifie les étapes suivantes.

Définition 9 (cohérence d'arc)
Une variable dans la portée d'une contrainte est arc-cohérente si chaque valeur de son domaine est contenue dans un support de la contrainte. Une contrainte est arc-cohérente si toutes les variables de sa portée sont arc-cohérentes.

Exemple 6
Considérons la contrainte $x < y$ où les variables ont toutes deux le domaine $\{1,2,3\}$. Comme x ne peut pas être affectée à la valeur 3, il n'y a pas de support pour cette contrainte quand $x = 3$, donc nous pouvons retirer la valeur 3 du domaine de x. De la même façon, y ne peut jamais être affectée à la valeur 1, donc il n'y a pas de support avec $y = 1$, et nous pouvons retirer la valeur 1 du domaine de y.

Si une variable n'est pas arc-cohérente avec une autre, elle peut le devenir en retirant certaines valeurs de son domaine. C'est la propagation de contraintes qui assure l'arc-cohérence : elle enlève du domaine de la variable chaque valeur qui ne correspond à aucune valeur de la seconde variable. Cette transformation conserve les solutions du problème puisque les valeurs retirées ne font partie d'aucune solution.

Nous allons décrire plusieurs algorithmes pour rendre un CSP arc-cohérent, AC1, AC3 et AC2001 ; dans nos travaux ce sont AC3 et AC2001 qui sont utilisés. Un algorithme naïf, AC1, parcourt toutes les paires de variables en rendant les contraintes arc-cohérentes. Ce parcours est répété jusqu'à ce qu'aucun domaine ne soit réduit pendant tout un parcours.

L'algorithme AC3 (voir l'algorithme 1) améliore cette idée en ignorant les contraintes qui n'ont pas été modifiées depuis leur dernière vérification. En pratique, on commence avec un ensemble contenant tous les couples contraintes et variables, la variable du couple n'étant pas affectée ; à chaque étape, on enlève un couple contrainte et variable de l'ensemble et on lui applique l'arc cohérence avec l'algorithme 2 *révise* ; si cette opération peut rendre arc-incohérente un couple contrainte et variable, on ajoute ce couple dans l'ensemble à vérifier. Ainsi un couple rendu arc-cohérent est ignoré jusqu'à ce que le domaine d'une des variables de la contrainte soit changé.

L'algorithme 2 *révise* se charge de rendre arc-cohérent une variable pour une contrainte donnée en cherchant des supports adéquats avec l'algorithme 3 *chercheSupport/AC3*. Ce dernier algorithme parcourt les tuples valides d'une contrainte, en fixant une variable donnée en entrée à une valeur aussi donnée en entrée, jusqu'à trouver un support.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Algorithme 1 : appliqueArcCohérence (P, A, Y)

Entrées : Un CSP $P = (X, D, C)$, une affectation partielle A, un ensemble de variables à vérifier Y
Sorties : $faux$ si P est arc-incohérent, $vrai$ sinon

```
1  Q ← ∅                                          // Q contient des contraintes à vérifier
2  pour chaque (c, x) ∈ C× portée(c) | x ∉ A ∧ ∃y ∈ portée(c) ∩Y , y ≠ x faire
3      Q ← Q ∪ {(c, x)}
4  tant que Q ≠ ∅ faire
5      choisir et supprimer (c, x) dans Q
6      si révise(c, x) alors                      // D(x) a été réduit
7          si D(x) = ∅ alors
8              retourner faux
9          pour chaque c' ∈ C | x ∈ portée(c') faire
10             pour chaque x' ∈ portée(c') | (c', x') ≠ (c, x) ∧ x' ∉ A faire
11                 Q ← Q ∪ {(c', x')}
12 retourner vrai
```

Algorithme 2 : révise (c, x)

Entrées : Une contrainte c, une variable x
Sorties : $vrai$ si le domaine de x est modifié, $faux$ sinon

```
1  nbÉléments ← |D(x)|
2  pour chaque v ∈D(x) faire
3      si ¬ chercheSupport(c, x, v) alors
4          D(x) ←D(x)\{v}
5  retourner nbÉléments ≠ |D(x)|
```

L'algorithme *chercheSupport/AC2001* (voir l'algorithme 4) améliore AC3 en conservant une trace de la dernière valeur compatible avec un triplet (c, x, v) d'une contrainte c, d'une variable x et d'une valeur v trouvées lors d'une mise à jour d'un domaine. Ceci est réalisé par le tableau *last* dans l'algorithme 4. Ce tableau est utilisé dans les lignes 1, 4 et 8. Il est mis à jour ligne 11.

Nous résumons les compromis des complexités temporelles et spatiales des algorithmes AC1, AC3 et AC2001 dans le tableau 2.1.

2.2.3 Stratégies de retour arrière

Nous présentons ici les différentes stratégies de retour arrière considérées pour nos travaux. Nous présenterons d'abord le retour arrière chronologique (*Chronological Back-Tracking* ou CBT). Nous verrons ensuite que CBT peut être facilement combiné avec des algorithmes de propagation de contraintes. Enfin, nous introduirons d'autres stratégies de retour arrière dynamiques complètes et enfin une stratégie incomplète.

Retour arrière chronologique (CBT) La méthode de la plus simple pour parcourir un arbre de recherche est le retour arrière chronologique (voir l'algorithme 5 et la fi-

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Algorithme 3 : chercheSupport/AC3 (c, x, v)

Entrées : Une contrainte c, une variable x, une valeur v
Sorties : $vrai$ si $\sup(c)_{x=v} \neq \emptyset$, $faux$ sinon

```
1  tuple ← premierTupleValide((c, x, v))
2  tant que tuple ≠ nil faire
3      si tuple ∈ rel(c) alors
4          retourner vrai
5      tuple ← tupleValideSuivant((c, x, v),tuple)
6  retourner faux
```

Algorithme 4 : chercheSupport/AC2001 (c, x, v)

Entrées : Une contrainte c, une variable x, une valeur v
Sorties : $vrai$ si $\sup(c)_{x=v} \neq \emptyset$, $faux$ sinon

```
1  si last[c, x, v] =nil alors
2      tuple ← premierTupleValide((c, x, v))
3  sinon
4      j ← premièrePositionInvalide(c,last[c, x, v])
5      si j = −1 alors
6          retourner vrai
7      sinon
8          tuple ←tupleValideSuivant((c, x, a),last[c, x, v],j)
9  tant que tuple ≠ nil faire
10     si tuple ∈rel(c) alors
11         last[c, x, v] ← tuple
12         retourner vrai
13     tuple ← tupleValideSuivant((c, x, v),tuple)
14 retourner faux
```

complexité	temporelle	spatiale
AC1	$\mathcal{O}(nm^3)$	$\mathcal{O}(1)$
AC3	$\mathcal{O}(nm^3)$	$\mathcal{O}(n)$
AC2001	$\mathcal{O}(nm^2)$	$\mathcal{O}(nm)$

Tableau 2.1 – *Complexités temporelles et spatiales des algorithmes AC1, AC3 et AC2001. Les complexités sont paramétrées par le nombre de variables n et la taille du plus grand domaine m.*

gure 2.3), qui construit de manière incrémentale une affectation tout en vérifiant à chaque étape si cette affectation est cohérente, c'est-à-dire respecte toutes les contraintes. Cela correspond à une construction de l'arbre de recherche en profondeur d'abord.

Nous appellerons *niveau* d'une variable x dans une affectation, le nombre de variables affectées au moment où x est affectée.

Le retour arrière chronologique est présenté en version récursive dans l'algorithme 5. La condition d'arrêt est, soit d'avoir trouvé une solution et dans ce cas la valeur $vrai$ est renvoyée, soit toutes les valeurs ont été testées pour la première variable et ont échoué, alors le CSP est prouvé incohérent et la valeur $faux$ est renvoyée. À chaque étape, cette

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

méthode choisit une variable et fait un appel récursif pour chaque valeur du domaine de cette variable, en ajoutant à l'affectation courante le couple variable et valeur désigné.

Cette méthode développe ainsi un arbre de recherche. Un nœud de l'arbre est une affectation d'une variable à une valeur et ses branches sont les tentatives pour les variables suivantes d'être affectées pour compléter l'affectation courante.

La complexité en temps de CBT est exponentielle en n, le nombre de variables. Elle souffre du grand nombre de redondances qu'elle génère. L'espace de recherche à explorer est immense : il est de l'ordre de d^n possibilités, d étant la taille maximale des domaines des variables. Notons que cette méthode peut, malgré un espace de recherche gigantesque, trouver une solution rapidement : l'arbre de recherche développé par le retour arrière chronologique sera petit si une solution est trouvée rapidement.

Algorithme 5 : RetourArrièreChronologique (P, A)

Entrées : Un CSP $P = (X, D, C)$, une affectation partielle cohérente A
Sorties : $vrai$ si P a une solution qui est un sur-ensemble de A, $faux$ sinon

```
si A est totale alors
    retourner vrai
sinon
    Choisir x_i ∈ X, x_i étant non affectée dans A, selon une heuristique donnée
    pour chaque v ∈ D(x_i) faire
        si A ∪ {(x_i, v_i)} est cohérente ∧ RetourArrièreChronologique(P, A ∪ {(x_i, v)}, P) alors
            retourner vrai
    retourner faux
```

La figure 2.3 illustre le comportement de CBT sur l'instance des 4-reines.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

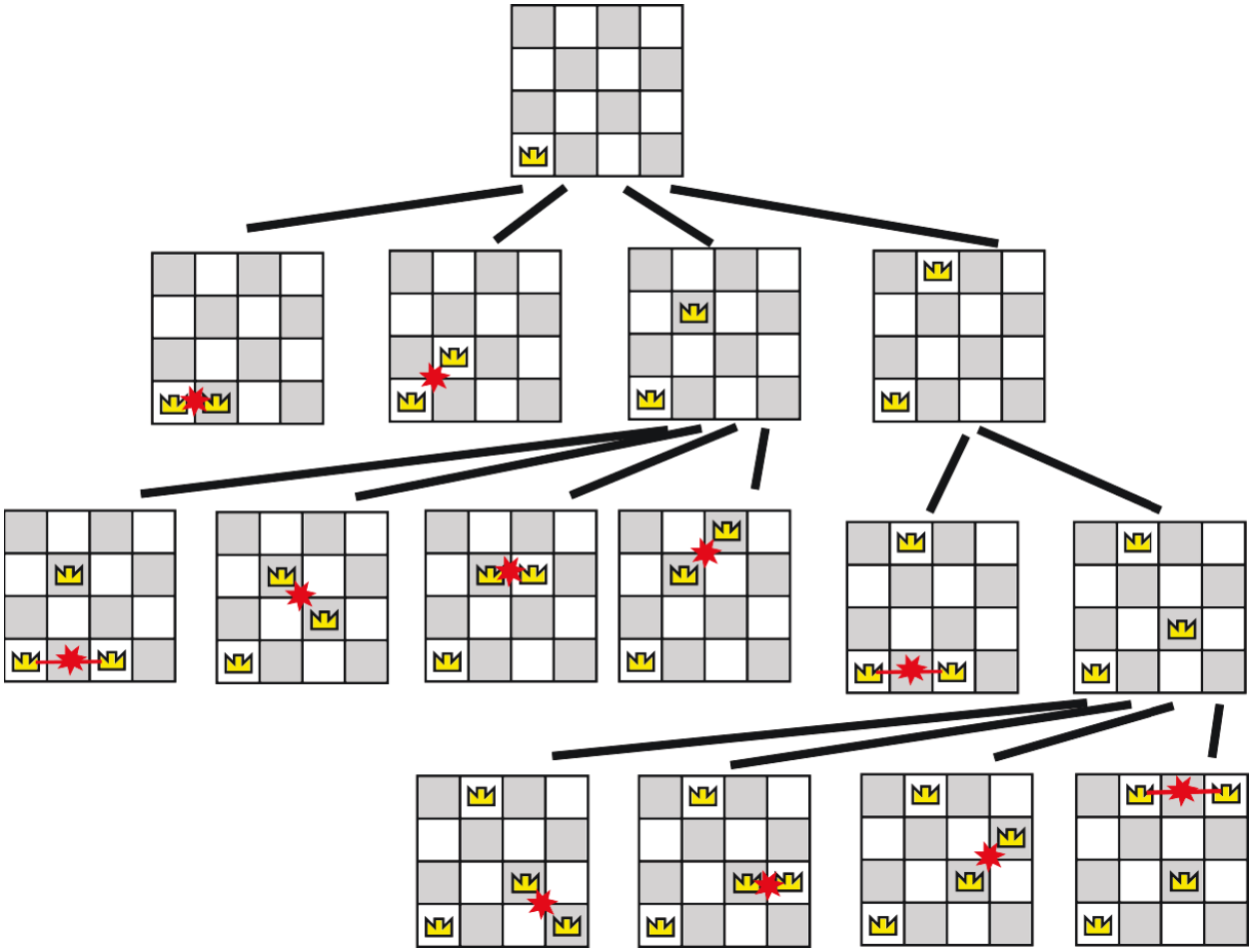


FIGURE 2.3 – *Comportement du retour arrière chronologique sur l'instance des 4-reines, douze premières étapes, essai de la position (1,1). Il manque neuf étapes avant de trouver une solution.*

Maintenir une cohérence lors de la résolution Pour réduire l'arbre de recherche construit par CBT, nous pouvons propager les contraintes après chaque affectation d'une valeur à une variable, entre les lignes 5 et 6 de l'algorithme 5. Notons que dans ce cas, les domaines doivent être restaurés à chaque retour en arrière. Ce point ne sera pas développé ici. Nous introduisons ici deux algorithmes qui maintiennent les cohérences locales introduites en section 2.2.2 pendant l'exécution de CBT : *Forward checking* et *Maintaining Arc Consistency*.

Pendant la résolution, l'algorithme de propagation de contraintes *Forward Checking (FC)* (voir figure 2.4) rend arc-cohérent chaque contrainte liant une variable nouvellement affectée et une variable non affectée. À chaque nouvelle affectation de variable pendant la recherche, FC vérifie le domaine des variables voisines et enlève les valeurs qui ne sont plus cohérentes depuis la nouvelle affectation. Ceci est fait en une seule passe car l'arc-cohérence d'une contrainte, une fois établie, n'est pas remise en cause.

La figure 2.4 présente le comportement de FC sur l'instance des 4-reines. Les ratures symbolisent le filtrage effectué par FC. Nous voyons que des échecs sont évités, par exemple dans la deuxième étape, la reine est placée en troisième position en partant du bas car la deuxième est déjà filtrée. Nous remarquons aussi que des échecs sont détectés en avance grâce au filtrage complet de domaine, comme par exemple dans la

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

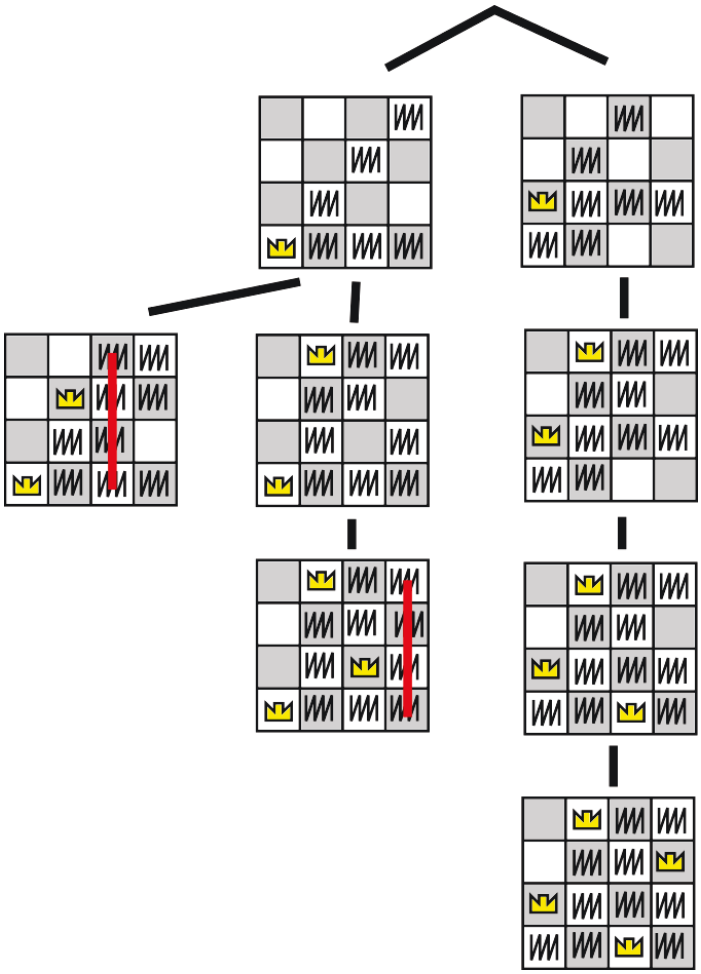


FIGURE 2.4 – *Comportement du Forward Checking sur l'instance des 4-reines, en huit étapes.*

quatrième étape où le domaine de la variable correspondant à la quatrième reine est vide. Comparé à la version sans filtrage en vingt et une étapes, CBT avec FC trouve une solution en seulement huit étapes.

De la même façon, un problème peut-être rendu et maintenu arc-cohérent par un algorithme appelé *Maintaining Arc Consistency (MAC)* consistant à appeler l'algorithme 1 en restreignant Q à l'ensemble des contraintes portant sur la variable affectée.

La figure 2.5 nous montre le comportement de MAC sur l'instance des 4-reines. Cela demande trois étapes de moins à MAC pour résoudre ce problème qu'à FC. En particulier, le filtrage plus puissant de MAC permet de détecter les échecs encore plus tôt, comme le montre la première étape où le domaine de la troisième variable est vidé dès le début par absence de support pour chacune des valeurs de cette variable. Les cinq étapes de MAC pour arriver à une solution sont à comparer avec les vingt et une étapes sans filtrage et les huit étapes pour FC.

Un point clé pour une résolution efficace consiste à trouver le bon compromis entre la puissance d'un filtrage et son coût en temps. Ce bon compromis peut être étudié de façon théorique en comparant les complexités en temps et les puissances des filtrages, mais son efficacité en pratique doit être évaluée expérimentalement et peut varier en fonction des instances considérées.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

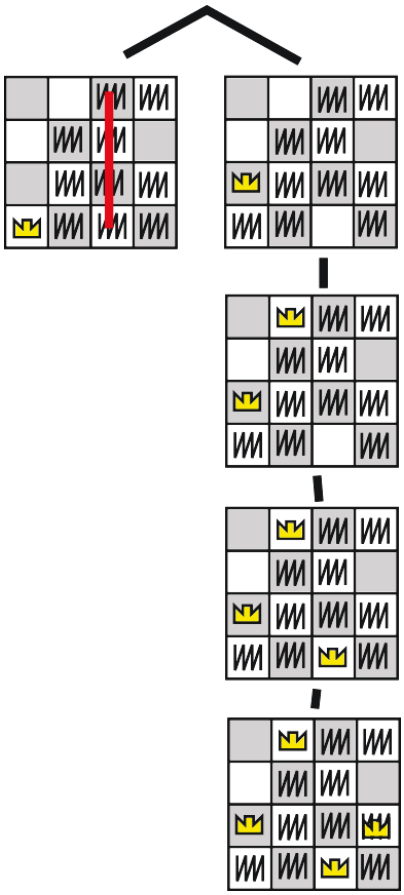


FIGURE 2.5 – *Comportement de Maintaining Arc Consistency sur l'instance des 4-reines, en cinq étapes.*

Retour arrière dirigé par les conflits (CBJ) En dehors de CBT, il existe d'autres stratégies pour explorer l'espace de recherche. L'objectif est de diminuer les redondances dans l'exploration en enregistrant des informations pertinentes au cours de la recherche.

Exemple 7
Considérons le CSP suivant :

- *il y a* n *variables, à savoir* $X = \{x_1, x_2, \ldots, x_n\}$
- *les domaines sont définis par*

$$\begin{aligned} D(x_1) &= \{n, n+1\} \\ D(x_i) &= \{i, i+1\}, \forall i \in \{2, \ldots, n-2\} \\ D(x_{n-1}) = D(x_n) &= \{n-1, n\} \end{aligned}$$

- *toutes les variables doivent avoir des valeurs différentes, c'est-à-dire*

$$\forall (x_i, x_j) \in X \times X, i \neq j \Rightarrow x_i \neq x_j$$

Supposons que CBT choisisse la prochaine variable à affecter dans l'ordre croissant de x_1 *jusqu'à* x_n *et qu'il choisisse les valeurs par ordre croissant également. Dans ce cas, il affecte* n *à* x_1 *en premier. Ensuite, pour chaque* $i \in \{2, \ldots, n-1\}$*, il affecte* i *à* x_i*. Enfin au moment où il affecte* x_{n-1} *à* $n-1$*, le domaine de* x_n *devient vide (car* $x_1 = n$*) et une incohérence est détectée.*

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Dans cet exemple, l'incohérence vient du premier choix $x_1 = n$ (et du choix pour x_{n-1} aussi). Cependant, au lieu de sauter directement à ce mauvais choix lors de l'échec, CBT essaie toutes les 2^{n-1} affectations possibles avec $x_1 = n$ avant de revenir sur une autre valeur pour x_1.

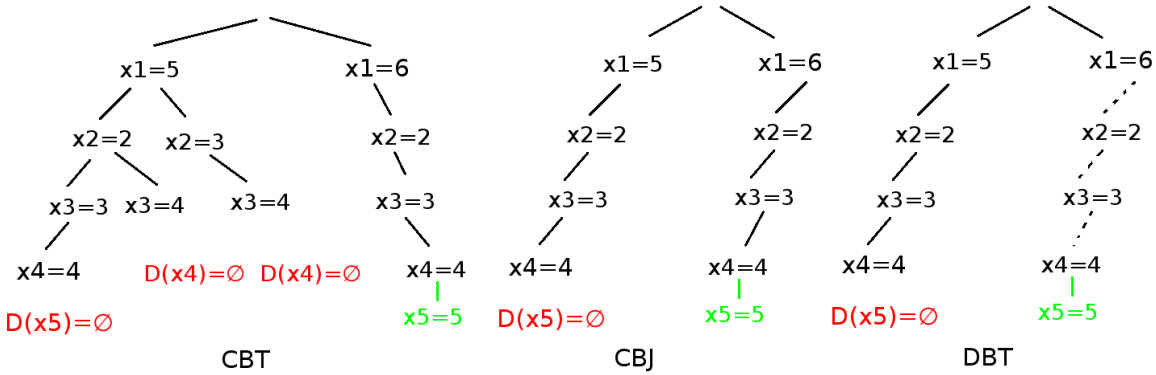


FIGURE 2.6 – *Arbres de recherche développés par CBT, CBJ et DBT pour le CSP de l'exemple 7 avec $n = 5$*

Nous illustrons avec la figure 2.6 le comportement de CBT pour $n = 5$. CBT désaffecte toute la partie gauche après l'échec en x_5, en essayant la valeur 4 pour x_3, puis la valeur 3 pour x_2, avant de revenir sur x_1 et de trouver la solution.

Dans le retour arrière dirigé par les conflits (*Conflict directed BackJumping* ou CBJ) chaque variable x a un ensemble de conflits $CS(x)$. Il contient l'ensemble des variables qui ont eu un échec lors d'un test de cohérence avec la valeur courante. Il faut maintenir cet ensemble à jour à chaque vérification de cohérence.

À chaque fois qu'une valeur est supprimée d'un domaine d'une variable, une explication est enregistrée contenant la raison de cette suppression. Plus précisément, une explication d'élimination d'une valeur v du domaine d'une variable x dû à l'affectation courante de variable $y_1, \ldots, y_k$ s'écrit :

$$expl(x \neq v) = \{y_1, \ldots, y_k\}$$

Une raison est nécessairement un ensemble (potentiellement vide) de variables affectées. Ainsi, en cas d'échec sur une variable (son domaine est devenu vide), les causes de cet échec sont obtenues en faisant l'union des variables contenues dans les explications des suppressions des valeurs de son domaine. Une explication de la suppression d'une valeur pourrait être conservée tout au long de la résolution permettant d'aller supprimer cette valeur dès lors qu'elle serait de nouveau contenue dans l'affectation courante. Toutefois, l'espace mémoire nécessaire à cela est prohibitif et est proportionnel à la taille de l'espace de recherche. De ce fait, une explication est conservée tant qu'elle est cohérente avec l'affectation courante, c'est à dire que toutes les variables qu'elle contient sont affectées. Elle est « oubliée » dès qu'une de ses variables est désaffectée et la valeur dont elle justifiait la suppression est restituée dans son domaine.

Il faut noter qu'il peut arriver qu'une valeur soit supprimée d'un domaine et que l'explication de cette suppression soit vide. Par exemple, si la valeur courante de la première variable affectée est supprimée de son domaine, l'explication de cette suppression est vide. Dans ce cas, la valeur est prouvée incohérente et peut être définitivement supprimée du problème.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Quand le domaine de la variable courante x devient vide, CBJ remonte directement à la dernière variable y affectée dans l'ensemble $CS(x)$. Toutes les variables intermédiaires entre x et y sont également désaffectées.

C'est le fait de potentiellement désaffecter les variables intermédiaires qui donne son intérêt à CBJ, car cet algorithme évite ainsi de déployer les sous-arbres liés aux variables intermédiaires.

Cette méthode évite assez directement beaucoup de redondances. Nous pouvons donc imaginer que sur des instances dont l'espace de recherche est trop grand pour être résolu uniquement avec CBT – même en y ajoutant la propagation des contraintes –, éviter ces redondances permet de pousser la résolution bien plus loin.

Dans la figure 2.6, lors de l'échec en x_5, nous avons :

$$CS(x_5) = \{expl(x_5 \neq 4) = \{x_4\}, expl(x_5 \neq 5) = \{x_1\}\}$$

donc nous désaffectons la variable x_4. Ici nous ajoutons à $CS(x_4)$ l'explication $expl(x_4 \neq 4) = \emptyset$, c'est-à-dire que la valeur 4 pour la variable x_4 est incohérente. Lors de l'échec suivant en x_4, nous avons :

$$CS(x_4) = \{expl(x_4 \neq 4) = \emptyset, expl(x_4 \neq 5) = \{x_1\}\}$$

nous pouvons donc immédiatement désaffecter x_3, x_2 et enfin x_1, pour affecter x_1 à 6.

Retour arrière dynamique (DBT) [Gin93] Le retour arrière dynamique (*Dynamic Backtracking* ou DBT) est construit sur la base de CBJ. Au lieu de désaffecter les variables intermédiaires lors d'un échec, DBT ne désaffecte qu'une variable, la dernière affectée étant coupable (on dira aussi en cause), c'est-à-dire contenue dans une explication d'élimination d'une valeur du domaine de la variable en échec. Comme les variables intermédiaires ne sont pas en cause dans l'échec courant, l'algorithme conserve leur affectation. Ceci indique que l'échec courant a lieu dans une partie du problème distincte de là où les variables intermédiaires sont. C'est donc pour utiliser la structure du problème que DBT garde les variables intermédiaires affectées. Ainsi, DBT peut chercher à résoudre plusieurs parties du problème sans interférences. Ce comportement est à comparer avec celui de CBJ qui déferait le travail réalisé sur une partie pour revenir à une autre partie du problème. Les preuves de correction et de complétude de DBT peuvent être trouvées dans [Gin93].

Notons que dû à son mécanisme, DBT ne structure pas l'espace de recherche en arbre mais en un graphe plus général.

Nous pouvons supposer que DBT se comporte bien sur des problèmes où la structure amène facilement l'exploration de l'espace de recherche sur une situation où plus d'une partie du problème est à résoudre en « même temps ».

Dans la figure 2.6, DBT, lors de l'échec en x_5, désaffecte x_1 en gardant les affectations de x_4, x_3 et x_2 (symbolisé par les pointillés dans la figure). Les ensembles de conflits sont les mêmes que pour CBJ.

Il faut noter qu'après une nouvelle affectation de la variable x_1, elle se trouve affectée après les variables x_2, x_3 et x_4. Dans le cas d'un échec futur dont les variables x_1, x_2, x_3 et x_4 seraient les causes, le retour à la dernière variable en cause nous conduirait à désaffecter à nouveau x_1. Notons donc que DBT modifie l'ordre d'affectation des variables imposé préalablement par l'heuristique de choix de variables (si l'heuristique est statique). L'article [Bak94] a montré que cela pouvait fortement nuire à ses performances.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

DBT avec ré-ordonnancement (CBJR) [ZSZM06] La méthode CBJR, appelée *Retroactive Ordering for Dynamic Backtracking* par ses auteurs, tente de conserver le comportement de DBT sans pour autant perturber l'heuristique de choix de variables [Bak94]. Lors d'une affectation réussie, CBJR remonte la variable nouvellement affectée tant que celle-ci n'est pas en conflit avec la variable avant elle dans l'affectation et que l'heuristique de choix de variable donne un score plus important à la variable à remonter. Lors d'un échec, CBJR a le comportement de CBJ – d'où le nom CBJ avec *ré-ordonnancement* – .

Plus précisément, lors d'une affectation réussie, CBJR remonte dans l'affectation la variable affectée. Cette opération est effectuée si la valeur de l'heuristique de choix de variable pour cette nouvelle variable est meilleure que celle de la variable précédente dans l'affectation et n'apparaît dans aucune explication avec la variable fraîchement affectée.

L'objectif de CBJR est de rapprocher les variables liées par des conflits de sorte à ce que les sauts en arrière de la méthode CBJ ne puissent conduire à désaffecter un grand nombre de variables. Ceci est réalisé tout en conservant l'impact de l'heuristique de choix de variables sur l'ordre d'affectation.

La figure 2.7 illustre le mécanisme de CBJR. Une affectation étant réussie pour x_5 avec la valeur e, CBJR remonte cette variable dans l'affectation au dessus de x_4 et x_3 qui ont des scores heuristiques plus faibles. La variable x_5 ne passe cependant pas au dessus de x_2 car cette dernière a participé au filtrage du domaine de x_5.

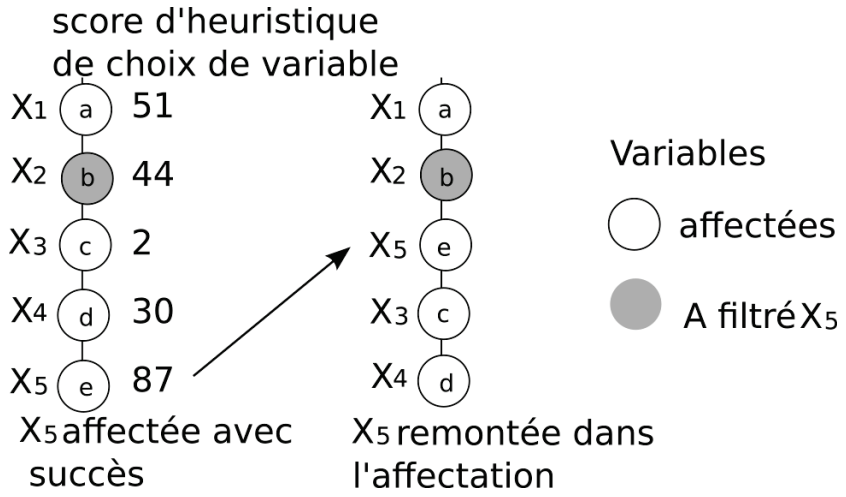


FIGURE 2.7 – *Comportement après une affectation réussie avec le réordonnancement*

Decision Repair (DR) L'algorithme *Decision Repair* (DR) [JL02] poursuit l'idée du DBT en permettant de ne désaffecter qu'une variable en cause lors d'un échec. Par contre, là où DBT désaffecte la dernière variable en cause dans l'échec courant, DR propose de choisir la variable à désaffecter parmi toutes celles qui causent l'échec courant. Ce faisant, DR n'est pas un algorithme complet et perd donc la garantie de trouver une solution s'il en existe une, ou de prouver l'incohérence du problème le cas échéant. Cependant, il faut noter que DR reste capable de prouver l'incohérence grâce à l'enregistrement d'explications d'éliminations (comme dans CBJ et DBT), simplement il n'y a plus de garantie de résolution. L'incohérence peut être prouvée s'il y a un échec et aucune variable n'est en cause.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

L'article [PV04] a proposé une heuristique de choix de la variable à désaffecter *Min-Destroy* dont le but est de conserver le plus d'explications possibles afin de construire une potentielle preuve d'incohérence. Sachant qu'une explication est « oubliée » dès lors qu'une de ses variables est désaffectée, il propose donc de choisir aléatoirement comme variable à désaffecter une parmi celles dans l'ensemble $CS(x)$ qui minimisent le nombre d'explications qui les contiennent. En choisissant ainsi la variable à désaffecter, nous avons la certitude de remettre dans les domaines des variables un minimum de valeurs.

Cette heuristique peut être améliorée en donnant un plus grand poids aux explications construites par retour arrière que celles construites par propagation de contraintes, car ce premier type d'explication demande généralement plus de recherche pour être produit. Un plus grand poids peut être donné aux explications plus concises. Nous pouvons aussi l'améliorer en propageant les contraintes même sur les variables affectées pour récolter des informations plus précises pour l'heuristique.

Par exemple, pour le problème des 4-reines, DR va – comme CBT – arriver à un échec avec l'affectation $\{(x_1, 1), (x_2, 4)$ (voir figure 2.3 pour les premières étapes). Là où CBT, CBJ, CBJR et DBT désaffectent d'abord x_2 avant de trouver une autre valeur à x_1, l'heuristique de désaffectation de DR peut choisir de désaffecter x_1 d'abord, laissant x_2 affectée à la valeur 4.

Cet algorithme DR, comme DBT, structure l'espace de recherche en un graphe qui n'est pas forcément un arbre de recherche.

Résumé La figure 2.8 illustre les comportements pour le retour-arrière de CBT, CBJ, DBT et DR.

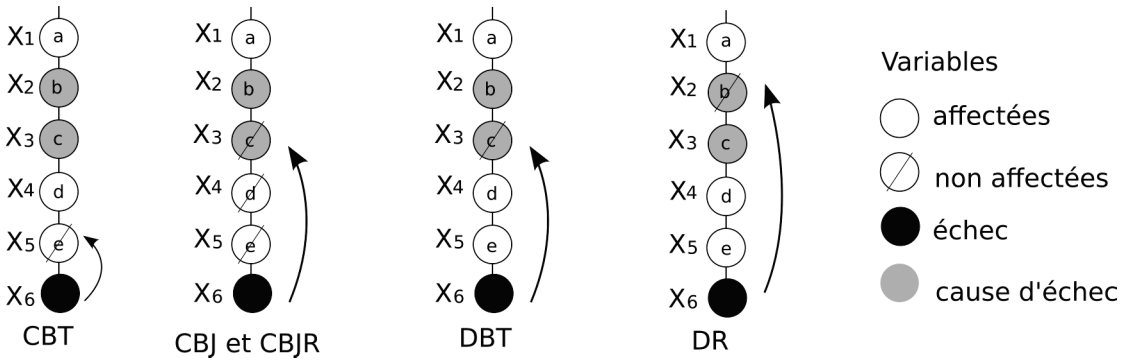


FIGURE 2.8 – *Comportement des différents algorithmes de retour arrière*

L'algorithme CBJ désaffecte les variables x_4 et x_5 qui ne participent pas à l'échec courant en x_6. Cela nous évite de déployer les sous-arbres, potentiellement larges, de x_4 et x_5.

DBT laisse affectées les variables x_4 et x_5 qui ne participent pas à l'échec courant en x_6. Ceci permet de continuer la recherche avec les valeurs courantes dans l'affectation de x_4 et x_5, évitant ainsi de jeter une partie des efforts de recherche sans lien avec l'échec courant.

La même figure illustre la capacité de DR à désaffecter une variable en cause dans l'échec courant qui n'est pas forcément la dernière variable en cause à avoir été affectée. Ici, x_2 est désaffectée (x_2 est en bien cause dans l'échec en x_6) alors que x_3, aussi en cause, a été affectée après x_2.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

2.2.4 Exploiter statiquement la structure du problème

Nous avons déjà abordé plusieurs techniques pour exploiter la structure d'un problème de façon dynamique. Les algorithmes CBJ, DBT, CBJR et DR sont guidés par les conflits qui sont mis en évidence pendant la recherche. Ces conflits sont dûs à la structure du problème.

D'autres approches utilisent la structure du problème pour assurer des garanties théoriques fortes sur la complexité en temps, mais aussi rendre plus efficace la résolution pratique [FQ85, DP87, DP89, GLS00, Dar01, JT03b, DM07]. Par exemple les travaux sur l'analyse de la micro-structure d'un CSP [MJT13] ou sur le graphe d'intersection des domaines [FLP14] exploitent directement la structure des CSP.

La décomposition arborescente Nous allons plus spécifiquement nous intéresser à la notion de décomposition arborescente. Dans le cas d'un CSP binaire où les contraintes lient au plus deux variables, la structure d'une instance est capturée notamment par le graphe des contraintes. La décomposition arborescente est un outil pour faire apparaître la structure d'un graphe.

Définition 10
[RS86] Une décomposition arborescente *d'un graphe $G = (S, A)$ est une paire (T, E), où $T = (N, F)$ est un arbre, et E est une fonction d'étiquetage associant à chaque sommet $p \in N$ un ensemble de sommets (appelé cluster) $E(p) \subseteq S$, tels que :*

- *chaque sommet de G apparaît dans au moins un cluster, $\forall v \in S, \exists p \in N, v \in E(p)$;*
- *Pour chaque arête de G, il existe au moins un cluster contenant les deux sommets de l'arête, $\forall (v, u) \in A, \exists p \in N, \{v, u\} \subseteq E(p)$;*
- *tous les clusters contenant un même sommet de S doivent être reliés par des chemins ne passant que par ces clusters, c'est-à-dire, pour tout sommet $v \in S$, l'ensemble $\{p \in N | v \in E(p)\}$ induit un sous-graphe connexe de T.*

La *largeur* d'une décomposition est la taille du plus grand ensemble $E(p)$ moins un. La *largeur d'arbre* ou *largeur arborescente* du graphe G est la plus petite parmi celles de toutes les décompositions possibles de G. Une décomposition arborescente est optimale si sa largeur correspond à la largeur d'arbre du graphe considéré. Calculer la largeur d'arbre d'un graphe est un problème NP-difficile.

Soit le CSP suivant :

- $X = \{A, B, C, D, E, F, G, H, I, J\}$
- $\forall x \in X, D(x) = \{1, 2, 3\}$
- $C = \{A < D, D = E, A \neq E, A - C = 1, A > B, C \neq J, C > F, B = F, F < I, G > B, G + 2 > H, B - 1 = H\}$

La figure 2.9 présente le graphe des contraintes de ce CSP et une décomposition arborescente de ce graphe.

Définition 11 (séparateur)
Étant donnée une décomposition arborescente (T, E) et une racine $r \in N$, le séparateur d'un sommet $i \in N \setminus \{r\}$ est l'intersection entre son cluster et le cluster de son père, c'est-à-dire, $E(i) \cap E(p_i)$ où p_i est le père de i dans l'arbre T enraciné en r.

Par exemple, pour la décomposition de la figure 2.9, si la racine est E_1, alors le séparateur de E_2 est $\{A\}$, le séparateur de E_3 est $\{B, C\}$,...

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

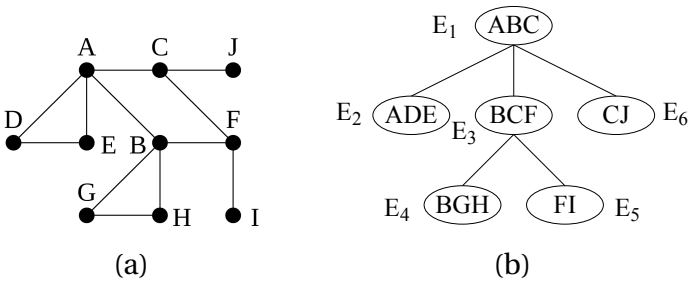


FIGURE 2.9 – *(a) Un graphe. (b) Une décomposition arborescente de ce graphe de largeur deux (figure issue de [JT03a]).*

Le problème de calculer une décomposition arborescente de largeur minimale est NP-difficile. Nous pourrions extraire toutes les cliques maximales du graphe mais il y en a un nombre potentiellement exponentiel. Nous utilisons l'algorithme heuristique décrit dans [JNT05] pour calculer une décomposition arborescente. Cet algorithme a une complexité en $\mathcal{O}(n^3)$. Il permet généralement de calculer une bonne décomposition (de faible largeur), sans aucun garantie sur la largeur cependant. Cet algorithme utilise la notion de graphe triangulé définie ci-dessous.

Définition 12 (graphe triangulé)
Un graphe est dit triangulé *si tous ses cycles de longueur supérieure ou égale à quatre ont une corde, c'est-à-dire une arête reliant deux sommets non consécutifs dans le cycle.*

Une propriété des graphes triangulés est qu'ils ont au plus n cliques maximales – n étant le nombre de sommets du graphe – et que ces cliques peuvent être extraites en temps polynomial. Rappelons qu'une clique maximale est un sous-ensemble de sommets C qui sont tous connectés deux à deux, et tel qu'il n'existe pas de sommet n'appartenant pas à C et connecté à tous les sommets de C.

L'algorithme de [JNT05] pour calculer une décomposition arborescente comporte les étapes suivantes :

1. Trianguler le graphe des contraintes. Cette étape consiste à ajouter des arêtes jusqu'à ce que le graphe soit triangulé. On utilise pour cela l'algorithme glouton *Min-Fill* [Kja90].
2. Extraire l'ensemble de toutes les cliques maximales du graphe triangulé.
3. Construire un graphe G dont chaque sommet correspond à une clique maximale, et tel que deux sommets soient reliés par une arête si leurs cliques associées partagent au moins un sommet. Pondérer chaque arête de ce graphe par la taille de l'intersection des cliques associées à ses sommets.
4. Calculer un arbre couvrant maximal de ce graphe, c'est-à-dire un arbre reliant tous les sommets du graphe et dont la somme des coûts des arêtes soit maximale.
5. Tant qu'il existe une arête (i, j) de l'arbre dont le coût est supérieur à une taille limite choisie, remplacer i et j par un nouveau sommet ij et associer à ij l'union des cliques associées à i et j. Mettre à jour en conséquence les arêtes (supprimer l'arête (i, j) et remplacer chaque arête (i, k) ou (j, k) par l'arête (ij, k)).

Dans l'exemple de la figure 2.9, le graphe n'est pas triangulé car il y a un cycle de longueur quatre sans corde (le cycle A,B,F,C). En ajoutant une arête CB, le graphe est

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

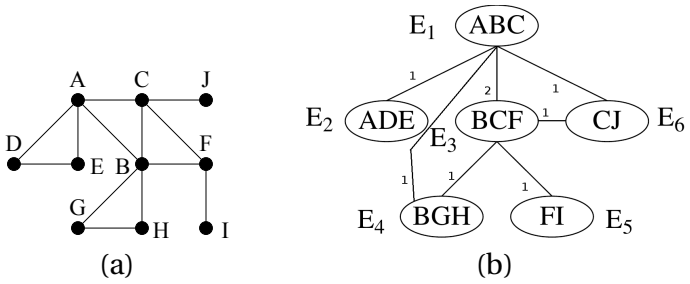


FIGURE 2.10 – *(a) Graphe triangulé. (b) Le graphe d'intersection des cliques maximales, avec le poids des arêtes (taille de l'intersection). (figure issue de [JT03a]).*

triangulé (voir figure 2.10). Les cliques maximales sont les clusters présents dans la partie droite de la figure 2.10. Le graphe d'intersection des clusters est présenté, avec la taille d'intersection entre clusters comme poids sur les arêtes. Nous donnons un arbre couvrant maximal à droite de la figure 2.9 (notez que l'arête entre E_1 et E_6 peut être remplacée par une arête entre E_3 et E_6 pour obtenir un autre arbre couvrant maximal). Cet arbre couvrant maximal est une décomposition arborescente du graphe initial.

Retour arrière avec décomposition arborescente (BTD) Une décomposition arborescente détecte l'indépendance entre différentes parties du problème. L'algorithme *Backtracking with Tree Decomposition* (BTD, voir algorithme 6), utilise cette structure afin de les résoudre séparément et donc plus efficacement. Cette résolution passe par un ordre d'affectation des variables compatibles avec la décomposition, mais aussi par l'enregistrement de la cohérence ou de l'incohérence des sous-problèmes, une fois que cela a été déterminé.

Illustrons l'indépendance de sous problèmes avec la figure 2.9. Si nous choisissons E_1 comme racine, alors une fois que les variables A, B et C sont affectées (le cluster E_1 entier est affecté), les trois sous problèmes enracinés en E_2, E_3 et E_6 sont indépendants :

— le sous problème enraciné en E_2 est le problème initial restreint aux variables D et E (leurs domaines étant potentiellement déjà réduits) ;
— de même en E_3 avec les variables C, F, G, H et I ;
— de même en E_6 avec la variable J.

Quand un sous problème est résolu, nous enregistrons un good (s'il est cohérent) ou un nogood (s'il est incohérent) correspondant aux valeurs de son séparateur. Par exemple quand E_3 est résolu, nous enregistrons un (no)good sur les valeurs de B et C ; si plus tard nous devons résoudre à nouveau ce sous problème avec ces mêmes valeurs, nous pourrons immédiatement utiliser le (no)good au lieu de le résoudre à nouveau.

Pour cela, BTD commence par choisir un cluster comme racine selon une heuristique à définir et construit de ce fait une arborescence de clusters, dans notre cas nous choisirons le cluster contenant le plus de variables comme racine. Les fils d'un cluster sont aussi ordonnés selon une heuristique à définir [JNT07a], dans notre cas nous ordonnerons les fils par taille d'intersection croissante avec le cluster père.

Définition 13 ((no)good structurel)
Un good (respectivement nogood) structurel est une affectation des variables d'un séparateur tel que le sous-problème associé à ce fils est cohérent (respectivement incohérent).

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

L'algorithme BTD est décrit dans l'algorithme 6. Nous avons choisi dans cet algorithme de présenter une version générique, qui peut être utilisée avec différents algorithmes de résolution de CSP (tels que ceux introduits à la section précédente). Nous supposons pour cela que nous disposons d'une méthode $buildModel()$, permettant de construire un modèle CP d'un CSP à résoudre, et d'une méthode $next()$, permettant de rechercher une solution au CSP (chaque appel à $next()$ retournant une nouvelle solution).

Cette fonction prend en entrée une instance CSP (X, D, C), une décomposition arborescente (T, E) et un sommet racine r de T. Notons que la décomposition arborescente (T, E) est une décomposition arborescente du CSP (X, D, C) lors du premier appel à BTD. En revanche, lors des appels suivants, (T, E) est un sous arbre de cette décomposition arborescente. BTD appelle tout d'abord la méthode *buildModel()* afin de construire le modèle CP, où l'ensemble des variables à affecter est restreint à celles du cluster racine E_r. Ensuite, elle appelle itérativement la méthode *next()* (ligne 2) pour rechercher des affectations cohérentes des variables de E_r, jusqu'à ce que soit une affectation puisse être étendue à une solution pour tous les sous-problèmes de r (de sorte que le problème est résolu – ligne 12), soit il n'existe plus d'autre affectation cohérente (de sorte que le problème est incohérent – ligne 13). Dans les lignes 3 à 10, BTD tente d'étendre l'affectation courante des variables de E_r à une solution pour tous les sous-problèmes : pour chaque fils i de r dans T, s'il existe un (no)good pour les valeurs affectées aux variables du séparateur $(E_r \cap E_i)$, alors ce sous-problème a déjà été résolu précédemment ; sinon BTD est récursivement appelée sur le sous-arbre de T enraciné en i, et le (no)good correspondant à ce sous-problème est enregistré. La boucle lignes 5 à 10 se termine soit quand un nogood est trouvé pour un fils (dans ce cas, la recherche passe à l'affectation suivante des variables de E_r), soit quand il existe un good pour tous les fils (dans ce cas, la recherche s'arrête sur un succès, ligne 12).

La réduction du nombre de redondances assure une complexité en temps de BTD exponentielle en la taille du plus gros cluster $\mathcal{O}(nd^{w+1})$ – où n est le nombre de variables, d la taille du plus grand domaine et w la largeur de la décomposition arborescente– , alors que sa complexité en espace est exponentielle en la taille maximum des séparateurs entre clusters $\mathcal{O}(nsd^s)$, où s est la taille du plus grand séparateur. Il est important donc de réduire la taille des séparateurs, ce qui motive la fusion de clusters (determinée par un paramètre à fixer), en cas de trop gros séparateur, lors du calcul de la décomposition arborescente.

2.2.5 Heuristiques de choix de variable

Nous avons vu dans la section 2.2.3 différents types de retour-arrière (CBT, CBJ, DBT, CBJR et DR). Quelle que soit la stratégie considérée, nous avons besoin d'une règle heuristique pour choisir la prochaine variable à instancier, parmi l'ensemble des variables qui ne sont pas encore affectées. Cette règle a un énorme impact sur les performances de résolution. Une idée classique est de provoquer les échecs le plus tôt possible dans la résolution afin de ne pas s'égarer sur une mauvaise piste, comme cela est illustré dans l'exemple 8.

Exemple 8 (changement de l'arbre de recherche selon l'ordre d'affectation des variables) *Soit un CSP à trois variables x_1, x_2, x_3 telles que $|D(x_1)| = |D(x_2)| = 4$ et $|D(x_3)| = 2$.*

Le nombre de nœuds internes change si l'ordre d'affectation change. Avec l'ordre $\langle x_1, x_2, x_3 \rangle$ il y a $4 * 4 = 16$ nœuds internes, mais avec l'ordre $\langle x_3, x_1, x_2 \rangle$ il y a $2 * 4 = 8$ nœuds internes ce qui, grâce au filtrage permet potentiellement d'élaguer une plus

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Algorithme 6 : BTD $((X, D, C), (K, T), r)$

Entrées : une instance CSP (X, D, C), une décomposition arborescente (K, T), et un sommet racine r de T
Sorties : Retourne *succès* s'il existe une fonction domaine $D' \subseteq D$ qui affecte de façon cohérente toutes les variables des clusters associés aux sommets de T ; retourne *échec* sinon.

```
1  model ← buildModel((X, D, C), K_r)
2  tant que model.next() ≠ null faire
3      Soit newD la fonction domaine retournée par next() (telle que newD affecte de
       façon cohérente toutes les variables de K_r)
4      Pour chaque fils i de r, soit A_i le tuple de valeurs affectées aux variables de
       K_r ∩ K_i dans newD
5      tant que il n'existe pas de fils j de r tel que A_j est un nogood
6      et il existe un fils i de r tel que A_i n'est pas un good faire
7          Soit T_i le sous-arbre de T enraciné en i
8          si BTD((X, newD, C), (K, T_i), i) = succès alors
9              enregistrer A_i comme good
10         sinon
11             enregistrer A_i comme nogood
12     si pour chaque fils i de r, A_i est un good alors
13         retourner succès
14 retourner échec
```

grande partie de l'arbre de recherche. Le nombre de feuilles de l'arbre reste constant quelle que soit l'heuristique de choix de variable.

Une heuristique de choix de variables classique est MinDomaine, qui choisit une variable qui a le plus petit domaine. Dans cette étude, nous en considérons deux améliorations connues MinDomaine sur degré dynamique [BR96] et MinDomaine sur degré pondéré [BHLS04].

MinDomaine sur degré dynamique MinDomaine sur degré dynamique (*Dynamic DEGree* ou ddeg), voir l'algorithme 7, choisit une variable x minimisant le ratio entre la taille de $D(x)$ et le nombre de variables non affectées partageant une contrainte avec x. En pratique, ddeg est calculé de manière incrémentale.

Algorithme 7 : MinDomaine sur degré dynamique (P, A)

Entrées : Un CSP $P = (X, D, C)$, une affectation partielle A
Sorties : une variable $x \in X$

```
1  pour chaque x ∈ X faire
2      ddeg[x] ← |{c ∈ C | x ∈ portée(c) ∧ y ∈ portée(c) ∧ x ≠ y ∧ y ∉ A}|
3  retourner argmin_{x∈X\vars(A)} |D(x)| / ddeg[x]
```


Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

MinDomaine sur degré pondéré MinDomaine sur degré pondéré (*Weighted DEGree* ou wdeg), voir l'algorithme 8, choisit une variable x minimisant le ratio entre la taille de $D(x)$ et la somme des poids des contraintes incluant x avec d'autres variables non affectées, où le poids d'une contrainte est le nombre d'échecs qu'elle a causés depuis le début de la recherche. Il faut ajouter pour FC comme pour MAC, à chaque fois qu'un domaine est vidé pendant le filtrage, l'instruction suivante :

$$poids[c] \leftarrow poids[c] + 1$$

Par exemple, dans l'algorithme 1 *appliqueArcCohérence*, il faut ajouter cette instruction entre la ligne 7 et 8.

Algorithme 8 : MinDomaine sur degré pondéré (P, A)

Entrées : Un CSP $P = (X, D, C)$, une affectation partielle A
Sorties : une variable $x \in X$

1 **pour chaque** $x \in X$ **faire**
2 $\quad \alpha_{wdeg}[x] \leftarrow \sum_{c \in C | x \in portée(c) \wedge (portée(c) \setminus \{x\}) \neq \emptyset} poids[c]$
3 **retourner** $\underset{x \in X \setminus vars(A)}{\operatorname{argmin}} \frac{|D(x)|}{\alpha_{wdeg}[x]}$

2.3 Discussion

Nous avons présenté dans ce chapitre un ensemble d'algorithmes pour résoudre des CSP. Nous avons vu qu'il y a différentes façons d'explorer l'espace de recherche, que ce soit par la stratégie de retour arrière, la méthode de propagation des contraintes ou encore l'heuristique de choix de variable. Toutes ces composantes permettent beaucoup de variantes, ce qui justifie la comparaison expérimentale pour analyser les performances et les complémentarités de ces différents mécanismes.

Nous avons également repéré différentes exploitations de la structure des CSP. Une exploitation dynamique est réalisée avec les retours arrières intelligents comme CBJ, CBJR, DBT et DR mais aussi avec les heuristiques de choix de variable ddeg et wdeg. Une méthode d'exploitation statique de la structure des CSP est l'algorithme BTD qui utilise une décomposition arborescente du graphe des contraintes du CSP.

Toutes ces méthodes sont complémentaires, c'est pour cela qu'il est pertinent de vouloir choisir dynamiquement la meilleure. C'est le sujet du prochain chapitre ou nous allons introduire des outils de l'apprentissage artificiel pour tenter de sélectionner automatiquement la meilleure configuration.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Chapitre 3

Programmation par optimisation

Sommaire

3.1 Apprentissage artificiel 34
3.1.1 Représentation des objets de l'apprentissage 34
3.1.2 Classification supervisée 34
3.1.3 Régression 36
3.1.4 Clustering 37
3.2 L'apprentissage pour la programmation par contraintes 38
3.2.1 Configuration 39
3.2.2 Portfolios 40
3.2.3 Sélection 40
3.2.4 Ordonnancement 41
3.2.5 Approches modifiant dynamiquement la stratégie de recherche 43
3.2.6 Apprentissage par renforcement pour les heuristiques 43
3.3 Discussion 43

Lors du développement d'un programme, nous faisons de nombreux choix de stratégies, d'heuristiques, de structures de données, ... qui peuvent avoir un impact important sur les performances. La programmation par optimisation [Hoo12] désigne le fait que l'on peut considérer tous ces choix comme des paramètres, plutôt que de faire des choix prématurés et utiliser des techniques d'optimisation pour choisir les meilleurs paramètres pour chaque contexte d'utilisation différent.

Nous avons vu au chapitre précédent que les algorithmes de résolution de CSP sont de bons exemples d'algorithmes pour lesquels nous avons différents choix possibles (la stratégie de retour arrière, les heuristiques d'ordres ou les algorithmes de filtrage) et nous verrons dans la deuxième partie de cette thèse que ces choix ont un impact fort sur les performances. Aussi proposons-nous dans la troisième partie de cette thèse d'utiliser la programmation par optimisation pour améliorer la résolution de CSP.

La programmation par optimisation utilise des techniques d'apprentissage artificiel que nous introduisons dans la section 3.1. Ces techniques ont déjà été largement utilisées pour améliorer les performances d'algorithmes de résolution de CSP et nous présentons les principales approches existantes dans la section 3.2.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

3.1 Apprentissage artificiel

L'apprentissage artificiel est un vaste domaine qui vise à permettre de développer des programmes capables d'apprendre. Des applications classiques sont la reconnaissance de formes, créer des comportements de joueurs pour des jeux tels que les échecs ou le go, la classification de données ou l'estimation de performances.

Voici deux ouvrages de référence en apprentissage artificiel : [CM02, BB13].

Nous pouvons distinguer deux types d'apprentissage : supervisé ou non supervisé. L'apprentissage supervisé dispose d'exemples pour apprendre un concept, par exemple reconnaître des photos avec ou sans chats, en donnant des exemples étiquetés. Dans le cas de l'apprentissage non supervisé, l'objectif est généralement de regrouper les éléments qui sont plus proches entre eux – ou alors semblent appartenir au même groupe – que d'autres éléments, on parle alors de *clustering*.

Nous nous limitons ici aux problèmes de classification, supervisée et non supervisée, et de régression qui sont utilisés en programmation par optimisation.

3.1.1 Représentation des objets de l'apprentissage

L'apprentissage s'appuie sur des données (des objets) qu'il faut représenter. Suivant le type de ces données, certaines représentations sont plus ou moins adaptées. Par ailleurs, toute description des données suppose déjà un pré-traitement, ne serait-ce que dans le choix des attributs de description ou la manière de faire face à des données imparfaites.

Nous allons supposer ici que les objets sont représentés par des vecteurs numériques de dimension d et nous appellerons $\mathcal{X} = \mathbb{R}^d$ l'espace de représentation des objets. Nous présenterons en section 3.2 des travaux concernant la représentation d'instances CSP par des vecteurs numériques (et en section 7.1.1 notre représentation dans cette thèse).

3.1.2 Classification supervisée

L'apprentissage d'une règle de classification est l'un des thèmes de l'apprentissage artificiel le plus traité. Il y a plusieurs raisons à cela : d'abord, on sait l'aborder du point de vue des théories de l'apprentissage, la plupart du temps dans le cas de deux classes (mais on peut assez facilement généraliser à un nombre quelconque). Ensuite, un grand nombre de méthodes et d'algorithmes existent, en particulier dans le cas où l'espace de représentation est numérique. On est alors dans le domaine classique de la reconnaissance statistique des formes (*statistical pattern recognition*). Enfin, apprendre à classer est un problème central de l'intelligence, naturelle comme artificielle.

Intuitivement, une règle de classification est un acte cognitif ou une procédure permettant d'affecter à un objet la famille à laquelle il appartient, autrement dit de le reconnaître.

La classification supervisée est une tâche classique en apprentissage artificiel. Il s'agit de classer des exemples donnés en entrée dans des classes connues à l'avance et pour lesquelles nous disposons d'exemples.

Définition 14 (exemple)
Soit X l'espace de représentation des objets, et $C = \{w_1, \ldots, w_k\}$ un ensemble fini de k classes. Un exemple est un couple $(x, u) \in X \times C$.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Un *ensemble d'apprentissage* permet à un *algorithme d'apprentissage* de construire une *règle de classification* en se basant sur des *exemples* déjà étiquetés.

Définition 15 (Ensemble d'apprentissage)
Ensemble d'exemples $A \subseteq \mathcal{X} \times \mathcal{C}$ *tel que* $\forall (x_i, w_i), (x_j, w_j) \in A, x_i = x_j \Rightarrow w_i = w_j$.

Un *ensemble de validation* permet à un modèle de classification de tester ses résultats en les comparant avec les étiquettes attendues en sortie, non fournies au modèle de classification.

Définition 16 (Ensemble de validation)
Ensemble d'exemples $V \subseteq \mathcal{X} \times \mathcal{C}$ *tel que* $\forall (x_i, w_i), (x_j, w_j) \in A, x_i = x_j \Rightarrow w_i = w_j$. *Un exemple de validation est un ensemble* $\{x_i \mid (x_i, w_i) \in V\}$.

Il faut que les ensembles de validation et d'apprentissage soient disjoints :

$$\forall (x_i, w_i) \in A, \forall (x_j, w_j) \in V, x_i \neq x_j$$

Il faut aussi que l'ensemble d'apprentissage soit représentatif de l'ensemble de validation, notamment dans le cas où les classes sont très déséquilibrées en proportion.

Définition 17 (Algorithme d'apprentissage)
Un algorithme d'apprentissage est un algorithme qui prend en entrée un ensemble d'apprentissage A *et renvoie une* règle de classification.

Définition 18 (Règle de classification)
Une règle de classification est un algorithme qui prend en entrée un exemple de validation et renvoie en sortie une classe.

Pour évaluer la qualité d'une règle de classification, nous pouvons compter le nombre d'exemples de l'ensemble de validation qui sont bien classés, c'est-à-dire le nombre de couples $(x_i, w_i) \in V$ tels que la règle de classification a retourné w_i pour l'entrée x_i. Une autre possibilité est d'étudier la *matrice de confusion*.

Définition 19 (Matrice de confusion)
La matrice de confusion d'une règle de classification est une matrice M *de taille* $\mathcal{C} \times \mathcal{C}$ *telle que* $M(w_i, w_j)$ *est le nombre d'exemples de l'ensemble de validation qui sont associés à la classe* w_i *dans l'ensemble de validation et à la classe* w_j *par la règle de classification.*
Le nombre d'exemples bien classés est $\sum_{w_i \in \mathcal{C}} M(w_i, w_i)$.

Une matrice de confusion est donnée dans le tableau 8.1, plus loin dans cet ouvrage.

Afin de tester la robustesse d'un algorithme de classification, il est intéressant de varier les ensembles d'apprentissage et de test plusieurs fois pour comparer les résultats. Une technique utile pour cela est la *validation croisée*.

Définition 20 (Validation croisée)
La validation croisée est une technique qui permet de générer plusieurs couples d'ensembles – d'apprentissage et de validation – à partir d'un seul ensemble d'exemples avec leurs étiquettes de classe.

La validation croisée à n *plis demande un entier* n *qui correspond au nombre de couples d'ensembles à créer. Pour créer ces ensembles, il suffit de partitionner aléatoirement notre ensemble initial en* n *parties. Chaque couple d'ensembles est ensuite obtenu en prenant une partie comme ensemble de validation et le reste des parties comme ensemble d'apprentissage.*

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Dans notre étude nous utiliserons le *leave-one-out* qui est une validation croisée où le nombre de plis est égal au nombre d'exemples, c'est-à-dire que l'ensemble de validation est un singleton et l'ensemble d'apprentissage contient tous les exemples sauf celui du singleton. Cela nous permet d'avoir le plus grand ensemble d'apprentissage, tout en garantissant la validité de l'expérimentation.

Maintenant que le problème de classification est bien défini, nous pouvons tenter de le résoudre.

Il existe de nombreux algorithmes de classification. Parmi les plus connus figurent :

— La méthode des k plus proches voisins (k *nearest neighbours* ou k-nn) [CH67] qui compare un vecteur à ses plus proches voisins selon une distance donnée. Pour estimer la sortie associée à un nouvel objet x, cette méthode prend en compte (de façon identique) les k exemples d'apprentissage les plus proches de la nouvelle entrée x. La classe de x sera la classe majoritaire parmi ses k plus proches voisins.

— L'algorithme C4.5 [Qui93] qui est un classifieur à base d'arbre de décision, l'idée étant de séparer le mieux possible les objets à chaque nœud de l'arbre.

— Les réseaux de neurones, qui permettent d'approximer une fonction en donnant des données en entrée à une série de fonctions de combinaison (les neurones), puis en apprenant de meilleurs paramètres pour ces fonctions, avec de l'apprentissage par renforcement par exemple.

— Les machines à support de vecteurs (SVM) [CV95], qui plongent les vecteurs dans un espace à plus de dimensions, afin de mieux séparer les vecteurs.

3.1.3 Régression

La régression est un ensemble de méthodes statistiques très utilisées pour analyser la relation d'une variable par rapport à une ou plusieurs autres.

La régression peut être vue comme un cas particulier de classification où l'ensemble des classes $\mathcal{C} = \mathbb{R}$. Ainsi, l'ensemble d'apprentissage A est inclus dans $\mathcal{X} \times \mathbb{R}$, de même pour l'ensemble de validation : $V \subseteq \mathcal{X} \times \mathbb{R}$.

Il s'agit dans ce cas d'utiliser A pour apprendre une fonction $f : \mathcal{X} \to \mathbb{R}$. La qualité de f est évaluée sur V, généralement en évaluant les écarts entre $f(x_i)$ et la valeur associée à x_i dans V.

Un modèle de régression classique est la régression linéaire. On y fait l'hypothèse que la fonction qui relie les objets à la sortie est linéaire dans ses paramètres. On appelle généralement modèle linéaire simple un modèle de régression linéaire avec une seule variable explicative. On a deux variables aléatoires, une variable expliquée Y, qui est un scalaire, une variable explicative X, également scalaire. On dispose de n réalisations de ces variables, $(x_i)_{1\leq i\leq n}$ et $(y_i)_{1\leq i\leq n}$. Le modèle a deux paramètres :

— β_0, l'ordonnée à l'origine ;

— β_1, le coefficient directeur ;

Soit $y_i = \beta_0 + \beta_1 x_i + u_i$ où u_i est le résidu ; chaque résidu lui-même est une réalisation d'une variable aléatoire U_i.

Nous pouvons faire une régression linéaire avec un *estimateur des moindres carrés ordinaires*. L'estimateur des moindres carrés ordinaires est la solution du programme de minimisation de la somme des carrés des écarts entre les valeurs prédites et les valeurs observées par rapport aux deux paramètres β_0 et β_1 :

$$\mathrm{S} = \mathrm{Argmin}_{\beta_0,\beta_1} \sum_{i=1}^{n} u_i^2 = \mathrm{Argmin}_{\beta_0,\beta_1} \sum_{i=1}^{n} (y_i - \beta_1 x_i - \beta_0)^2$$

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

On a :

$$\hat{\beta}_1 = \frac{\sum x_i \sum y_i - n \sum x_i y_i}{(\sum x_i)^2 - n \sum x_i^2} = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sum (x_i - \bar{x})^2}$$

$$\hat{\beta}_0 = \frac{\sum y_i - \hat{\beta}_1 \sum x_i}{n} = \bar{y} - \hat{\beta}_1 \bar{x}$$

avec $\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$ la moyenne empirique des x_i et $\bar{y} = \frac{1}{n} \sum_{i=1}^{n} y_i$ la moyenne empirique des y_i.

Notons que la régression linéaire peut servir d'outil pour un algorithme de classification. Les arbres modèles (*model trees*) sont un type d'arbre de décision avec des fonctions de régressions linéaires à leurs feuilles. Ils peuvent être utilisés pour des problèmes de classification en transformant un problème de classification en un problème d'approximation de fonction [FWI+98].

3.1.4 Clustering

Le problème de clustering cherche une partition $\Pi = (S_1, \ldots, S_k)$ d'un ensemble fini d'objets S. Les parties $S_1, \ldots, S_k$ sont généralement appelées clusters. Étant donnée une mesure de dissimilarité $d : S \times S \to \mathbb{R}$ telle que $d(x_i, x_j)$ est d'autant plus grande que les objets x_i et x_j sont dissimilaires, l'objectif est de trouver une partition regroupant des objets similaires dans un même cluster et répartissant les objets dissimilaires dans des clusters différents (les clusters sont alors bien séparés).

Différents critères peuvent être considérés pour évaluer la qualité d'une partition. Pour évaluer l'homogénéité d'un cluster, on peut calculer son diamètre, c'est-à-dire la dissimilarité maximum entre deux objets du cluster. Pour évaluer la séparation entre un cluster et les autres, on peut calculer sa coupure –c'est-à-dire la somme des dissimilarités entre les objets de ce cluster et des objets des autres clusters–, ou encore sa séparation (*split*) –c'est-à-dire la dissimilarité minimum entre un objet de ce cluster et un objet d'un autre cluster.

Il existe un grand nombre d'algorithmes de clustering. Un des plus connus est l'algorithme k-*means* [Har75], où k est un paramètre donné en entrée fixant le nombre de clusters dans la partition à trouver.

L'algorithme est initialisé avec k centroïdes (la moyenne des partitions) $m_1, \ldots, m_k$ (par exemple choisis aléatoirement). Ensuite il faut répéter jusqu'à convergence les étapes suivantes :

— assigner chaque objet à la partition la plus proche

$$S_i^{(t)} = \left\{ \mathbf{x}_j : \left\| \mathbf{x}_j - \mathbf{m}_i^{(t)} \right\| \leq \left\| \mathbf{x}_j - \mathbf{m}_{i^*}^{(t)} \right\| \forall\, i^* \in \{1, \ldots, k\} \right\}$$

— mettre à jour les centroïdes

$$\mathbf{m}_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{\mathbf{x}_j \in S_i^{(t)}} \mathbf{x}_j$$

La convergence est atteinte quand il n'y a plus de changements.

Comme le choix du paramètre k est déterminant dans k-means, une méthode à été proposée pour apprendre ce paramètre, il s'agit de G-means [HE04]. L'algorithme G-means (voir l'algorithme 9) est basé sur un test statistique pour l'hypothèse qu'un sous-ensemble des données suit une distribution gaussienne. G-means lance k-means en incrémentant k de façon hiérarchique jusqu'à ce que le test accepte l'hypothèse que

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

les données affectées à chaque cluster sont gaussiennes. De plus G-means ne requiert qu'un paramètre intuitif, le niveau de signification standard statistique α.

Algorithme 9 : G-means (X, α)

Entrées : Un ensemble d'objets X, un niveau de signification standard statistique α

Sorties : Un ensemble de centroïdes C

```
1 Soit C le singleton contenant la moyenne de X
2 k ← 1
3 tant que des centroïdes sont ajoutés faire
4     C ← k-means(C, X, k)
5     pour chaque centroïde c_j de C faire
6         si un test statistique montre que le cluster associé à c_j ne suit pas une
          distribution gaussienne (avec un niveau de confiance α) alors
7             remplacer c_j par deux centroïdes
8             k ← k + 1
```

3.2 L'apprentissage pour la programmation par contraintes

Les différentes techniques que nous venons d'introduire sont utilisables en programmation par contraintes. Plusieurs buts sont envisageables :

— problème de configuration : étant donnés un ensemble d'instances et une méthode de résolution comportant un certain nombre de paramètres, déterminer le meilleur paramétrage pour l'ensemble des instances ;
— problème de sélection : étant données une instance et une méthode de résolution comportant un certain nombre de paramètres, déterminer le meilleur paramétrage pour cette instance ;
— pendant la résolution, ajuster les paramètres de la méthode de résolution, ou apprendre des propriétés de l'instance pour mieux la résoudre.

Chacun de ces buts a un problème qui lui correspond. Nous allons présenter des méthodes de configuration, de création de portfolios, de sélection d'algorithmes et d'ordonnancement, puis la modification dynamique de la stratégie de recherche.

Différents types de paramètres peuvent être appris : des valeurs numériques (fixant des seuils, des poids, des fréquences, ...) changeant les performances de l'algorithme, des choix de conceptions ou de programmation (aussi appelés hyper-paramètres) qui changent l'algorithme lui-même ou encore le choix d'un algorithme parmi d'autres.

Dans un problème d'optimisation de paramètres, nous aurons en entrée un algorithme A paramétré par un ensemble de paramètres P ; pour chaque paramètre $p_i \in P$, l'ensemble des valeurs possibles pour p_i ; un ensemble E d'instances d'apprentissage pouvant être résolues par A ; et un indicateur de performances I permettant d'évaluer la qualité de A sur une instance de E (par exemple le temps de résolution, le nombre de nœuds développés par l'arbre de recherche, ...).

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

3.2.1 Configuration

Souvent le comportement d'un algorithme peut être amélioré en ajustant ses paramètres. Mais même les intuitions des experts pour ce qui est « correct » peuvent être fausses et nous voudrions éviter de tester à la main toutes les combinaisons.

Il est intéressant de se pencher sur la configuration des paramètres des solveurs. La configuration a pour objectif de choisir les bons paramètres pour une méthode de résolution, étant donné une classe entière d'instances à résoudre. Nous cherchons donc les valeurs de P optimisant A pour l'indicateur de performances I sur les instances de E.

Une hypothèse forte est nécessaire sur les instances à résoudre : elles doivent avoir des caractéristiques similaires. Si ce n'est pas le cas il faut diviser l'ensemble d'instances en sous-ensembles homogènes avec du clustering, puis appliquer la configuration automatique sur chaque cluster.

Le nombre d'instances d'apprentissage doit aussi être ni trop petit (au risque de sur-apprendre les paramètres), ni trop grand (car le processus de configuration devient alors lent).

Pour explorer l'espace de recherche offert par les paramètres, il faut échantillonner les valeurs des paramètres (l'exploration exhaustive est généralement trop coûteuse). L'échantillonnage est préférablement fait de manière aléatoire afin de considérer une plus grande diversité de configurations, voir la figure 3.1 (tirée d'une présentation de Lars Kotthoff à IJCAI 2013) pour une illustration. En échantillonnant de façon régulière, si le nombre total de configurations que nous pouvons tester est égal à neuf alors seules trois valeurs de chacun des deux paramètres seront testées. En revanche, en choisissant les valeurs des deux paramètres de façon aléatoire, un plus grand nombre de valeurs de chacun des deux paramètres pourra être testé, augmentant ainsi la probabilité d'évaluer une valeur pertinente.

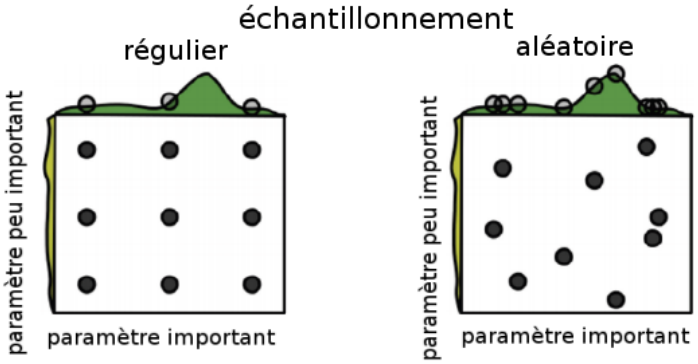


FIGURE 3.1 – *Exemple d'échantillonage régulier et aléatoire pour deux paramètres. Les courbes en couleurs en haut et à gauche des carrés montrent l'impact d'un paramètre sur la fonction objectif.*

Parmi les premiers succès dans la configuration de solveurs pour résoudre des problèmes combinatoires figurent F-race [BSPV02] et paramILS [HHLBS09]. Cette dernière méthode utilise un algorithme de recherche locale itérée pour chercher des paramètres dans l'espace des paramètres possibles :

1. initialiser les paramètres aléatoirement ;
2. tenter des modifications, un paramètre à la fois, en acceptant les nouvelles configurations quand elles améliorent les performances ;

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

3. répéter l'étape 2 jusqu'à ce qu'il n'y ait plus d'améliorations possible en changeant un seul paramètre à la fois.

3.2.2 Portfolios

Pour mieux résoudre un problème, une possibilité est de constituer un ensemble de méthodes de résolution. Cet ensemble de méthodes porte le nom de *portfolio*. L'idée vient de la gestion économique : nous souhaitons minimiser le risque en le répartissant sur plusieurs investissements. Dans le cas de problèmes combinatoires nous voudrions réduire le risque qu'un algorithme ait de mauvaises performances. Cela peut aussi être l'inverse : maximiser les performances de l'algorithme.

Si tous les algorithmes d'un portfolio se comportent de la même façon, le portfolio ne présente aucun intérêt. Idéalement, les algorithmes constituant un portfolio sont complémentaires.

Souvent, les portfolios sont simplement constitués de vainqueurs de compétitions. Cependant, il est possible de construire des portfolios contenant des algorithmes complémentaires [KMST10, XHLB10]. Une autre piste intéressante est d'analyser les contributions marginales de chaque algorithme [XHHL12] afin de ne choisir que les algorithmes apportant une contribution. La construction du portfolio est ainsi un point clé de sa performance.

L'approche présentée dans *Algorithm Portfolios* [GS01] nous apprend qu'il est intéressant d'avoir des algorithmes « risqués », c'est-à-dire, ayant une faible chance d'avoir des résultats excellents. Si nous disposons de plusieurs processeurs, lancer ces algorithmes risqués est une bonne méthode, si nous n'avons qu'un seul processeur, alors effectuer des *restart* avec une méthode aléatoire est également une bonne méthode.

3.2.3 Sélection

Parfois nous disposons de beaucoup d'approches pour résoudre un problème, mais nous savons qu'il n'y a pas d'algorithme parfait sur toutes les instances d'un problème. Comment alors choisir la meilleure approche pour une instance en particulier ?

Le problème de sélection [Ric76] est le suivant : étant donné une instance et un ensemble de méthodes de résolution, quelle méthode de résolution choisir ? Autrement dit, nous voulons construire un modèle pour choisir les valeurs de P pour chaque nouvelle instance.

Les étapes à suivre pour le problème de sélection sont les suivantes :
— extraire des propriétés de l'instance à résoudre
— choisir un algorithme grâce à ces propriétés
— mesurer les performances de l'algorithme choisi.

Cette fois-ci l'ensemble d'apprentissage devrait être grand, varié et représentatif de chaque catégorie d'instances du problème.

La caractérisation des instances par des propriétés est importante, car c'est en se basant sur ces propriétés que la sélection à lieu. Les propriétés classiques sont statiques ou dynamiques. Comme propriétés statiques, collectées par analyse de l'instance, nous avons généralement :
— nombre de variables, de valeurs, de contraintes, ... (minimum, maximum, moyenne, médiane, ...),
— analyse du graphe des contraintes (diamètre du graphe, densité des contraintes, ...).

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Comme propriétés dynamiques, collectées lors de sondages (*probing*), nous pouvons avoir :

— nombre de nœuds explorés à différentes limites de temps,
— profondeur maximum de l'arbre de recherche,
— nombre de valeurs filtrées à chaque nœud.

ISAC [KMST10]

La méthode « configuration d'algorithme par instance » (ISAC) utilise GGA [AST09], un algorithme génétique imposant différentes pressions de sélection sur différentes populations d'individus.

ISAC commence par classer les instances de l'ensemble d'apprentissage grâce à un algorithme non-supervisé qui détermine automatiquement le nombre de classes à trouver : G-means [HE04] (voir algorithme 9). Pour chaque classe d'instances trouvée, ISAC utilise GGA pour déterminer le meilleur paramétrage. Ensuite, pour chaque nouvelle instance à résoudre, ISAC recherche la classe d'instances la plus proche, et applique le paramétrage de cette classe sur cette instance. Cette approche mêle donc configuration et sélection.

3.2.4 Ordonnancement

Le problème d'ordonnancement vise à créer une liste de méthodes de résolution ainsi que de leurs temps d'exécution maximum pour résoudre un problème donné. Il s'agit donc de créer un « planning » pour les algorithmes à lancer.

Plusieurs méthodes existent pour tenter de résoudre ce problème d'ordonnancement. Nous présentons ici SATzilla, CPhydra et 3S.

SATzilla

Un problème de satisfaction de formules booléennes (SAT) est un CSP dont les variables ont un domaine contenant deux valeurs de vérité et dont les contraintes sont exprimées par des clauses logiques. SATzilla [XHHLB08] est une méthode d'ordonnancement qui a remporté de nombreuses médailles dans les compétitions de solveurs SAT. La méthode fonctionne avec les étapes suivantes :

1. Choisir un ensemble d'instances SAT, appelées instances d'apprentissage.
2. Constituer un portfolio de solveurs SAT candidats.
3. Identifier des propriétés caractérisant les instances. Ceci est en général accompli par un expert et non pas automatiquement. Pour être utiles, ces propriétés devraient être liées à la difficulté des instances et relativement aisées à calculer.
4. Pour chaque instance de l'ensemble d'apprentissage, calculer ces propriétés et exécuter chaque solveur du portfolio pour connaître son temps d'exécution.
5. Trouver un ou plusieurs solveurs à utiliser pour « pré-résoudre » les instances. Ces pré-solveurs seront lancés à l'étape 9 pour un court instant avant de calculer les propriétés, afin d'assurer de bonnes performances sur les instances faciles et de se concentrer uniquement sur les instances plus difficiles.
6. Trouver un solveur ayant les meilleures performances pour toutes les instances de l'ensemble d'apprentissage non résolues par les pré-solveurs et sur lequel le

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

calcul des propriétés est trop long. Ce solveur sera le solveur de secours (voir étape n°10).

7. Construire un modèle empirique de difficulté pour chaque solveur dans le portfolio, qui prédit le temps d'exécution du solveur sur chaque instance, en se basant sur les propriétés de l'instance (c'est une méthode de régression).
8. Choisir le meilleur sous-ensemble de solveurs à utiliser dans le portfolio final. C'est un problème de sélection de sous-ensemble : choisir l'ensemble de solveurs dont le portfolio correspondant a les meilleures performances sur l'ensemble de validation.
9. **Étant donnée une nouvelle instance à résoudre :** lancer chaque pré-solveur pour un certain temps prédéterminé.
10. Calculer les propriétés de l'instance. Si cela est trop long, passer la main au solveur de secours.
11. Sinon, prédire le temps d'exécution de chaque algorithme en utilisant le modèle empirique de difficulté calculé à l'étape 7.
12. Lancer les solveurs dans l'ordre, en partant de la meilleure prédiction et en les arrêtant à leurs temps limite ; jusqu'à ce que l'instance soit résolue.

L'efficacité de SATzilla réside dans sa capacité à apprendre des modèles de difficultés empiriques pouvant prédire avec précision le temps d'exécution d'un solveur sur une instance donnée, en utilisant des propriétés aisément calculables de cette instance. Ceci a été confirmé par les résultats que SATzilla a obtenu à plusieurs compétitions : SATzilla a obtenu cinq médailles en 2007 et 2009 à la *SAT competition* et a gagné le *SAT challenge* en 2012.

CPhydra [OHH+08]

La première méthode obtenant un grand succès avec de l'ordonnancement pour un portfolio pour les CSP (plus expressifs que SAT) est CPhydra [OHH+08]. Cette méthode utilise le raisonnement à partir de cas. Elle est composée de quatre étapes principales : récupération, réutilisation, révision et rétention.

Pour commencer nous calculons les propriétés d'une instance. À partir de ces propriétés, il faut chercher les cas les plus similaires à l'instance courante (avec l'algorithme K-nn).

La réutilisation des informations trouvées à l'étape précédente permet de calculer un ordonnancement des algorithmes performants sur les plus proches voisins du cas courant.

Suit la phase de révision, pendant laquelle l'ordonnancement est évalué et validé. Si possible, il est souhaitable à cette étape d'ajouter dans la base de cas celui qui vient d'être créé.

Enfin, pour la rétention, il s'agit d'exécuter à posteriori chaque algorithme du portfolio pour améliorer la base de cas.

Sélecteur de solveur SAT (3S) [KMS+11]

Pour palier à un potentiel manque de passage à l'échelle de CPhydra dû aux lourds calculs d'ordonnancement à chaque nouvelle instance, nous pouvons utiliser l'idée de 3S. Cette autre méthode est composée de deux parties : un ordonnancement statique

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

de solveurs (pendant 10% du temps alloué) et un choix dynamique d'un seul solveur (pour les 90% de temps alloué restant). L'ordonnancement statique est obtenu de manière similaire à l'ordonnancement de CPhydra, mais n'est calculé qu'une fois durant la phase d'entraînement. Si après l'exécution de cet ordonnancement, une instance présentée n'est pas résolue, alors le solveur à lancer est choisi parmi ceux qui résolvent la majorité des k plus proches cas connus de l'ensemble d'entraînement.

D'après [AGM13] 3S et SATzilla ont d'excellentes performances sur des CSP. Le temps de construction d'un modèle est cependant potentiellement très long pour SATzilla s'il y a beaucoup de solveurs dans le portfolio.

3.2.5 Approches modifiant dynamiquement la stratégie de recherche

Il est possible d'intervenir sur la stratégie de recherche dynamiquement durant la résolution d'un problème. Par exemple, dans [ESWR96] les auteurs proposent de changer la méthode de propagation des contraintes entre AC et FC selon certains paramètres, durant la recherche. Il devient possible de conserver l'arc-cohérence du problème en utilisant parfois moins de ressources pour y arriver – lorsque FC suffit et que cela est détecté.

Une autre possibilité d'apprentissage dynamique porte sur l'heuristique pour créer les branches de l'arbre de recherche. Cette idée a été explorée avec « approche dynamique pour changer d'heuristique » (DASH) [LKLM13]. Pour choisir l'heuristique à utiliser, l'algorithme DASH choisit, à certains intervalles de profondeur dans l'arbre de recherche, une nouvelle heuristique en fonction de propriétés du sous-problème restant à résoudre.

3.2.6 Apprentissage par renforcement pour les heuristiques

Nous avons déjà présenté l'heuristique wdeg en section 2.2.5. C'est une heuristique de choix de variable très efficace, qui tire sa force de sa simplicité : apprendre où se trouvent les conflits pour tenter de les résoudre en premier.

Il existe d'autres heuristiques de choix de variable faisant appel à des techniques d'apprentissage. Nous pouvons par exemple, comme dans la recherche « Activity », chercher les variables qui sont les plus actives au sens du nombre de fois ou leur domaine est réduit [MVH11].

Une autre piste consiste à estimer l'impact [Ref04] qu'aura une affectation de variable en terme de propagation, de nombre de valeurs éliminées. En maximisant l'impact de la variable à affecter nous espérons réduire le plus possible l'espace de recherche.

3.3 Discussion

Nous avons maintenant présenté un ensemble de techniques utilisant des méthodes provenant de l'apprentissage artificiel. La constitution d'un portfolio nous est d'un intérêt particulier pour la présentation de la suite de nos travaux. Nous avons vu que plusieurs méthodes sont envisagées lorsque nous disposons de plusieurs méthodes de résolution pour résoudre différentes instances d'un même type de problèmes. La sélection d'une méthode pour résoudre une instance donnée est une possibilité, mais aussi la configuration des paramètres d'une méthode.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

D'autres possibilités incluent la modification de la stratégie de résolution pendant la recherche, ce qui nécessite un apprentissage pour déterminer comment modifier cette stratégie. Les heuristiques de choix de variables sont aussi un élément qui peut s'affiner durant la recherche que ce soit en se concentrant sur les échecs, l'activité ou l'impact d'une variable.

Nous pouvons aussi mentionner l'apprentissage de contraintes durant la recherche, avec l'exemple de l'apprentissage de contraintes globales implicites [BCP+07].

Bien souvent l'aspect non déterministe des algorithmes est ignoré dans les approches de sélection et d'ordonnancement (ce n'est pas le cas pour les approches de configuration comme F-race). Quand on évalue les performances d'un algorithme sur une instance d'apprentissage, nous ne considérons qu'une seule exécution et nous considérons que la meilleure approche est celle qui a été la plus rapide pour résoudre le problème (ou bien celle qui a trouvé la meilleure solution dans le cas de problèmes d'optimisation). Une contribution de la thèse est d'expliciter le non déterminisme des algorithmes (en départageant les ex-aequo de façon aléatoire, au lieu d'utiliser des règles arbitraires comme l'ordre d'apparition des valeurs), et d'exécuter plusieurs fois chaque algorithme sur chaque instance d'apprentissage afin de pouvoir utiliser des tests statistiques pour déterminer si un algorithme est significativement meilleur qu'un autre pour une instance.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Deuxième partie

Un cadre générique pour les CSP

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Chapitre 4

Solveur générique

Sommaire

4.1 Présentation du solveur générique 47
4.2 Rappel de l'algorithme de [LBH04] 48
4.3 Extensions de l'algorithme à de nouvelles stratégies dynamiques . . 53
4.3.1 Inclure Decision Repair 54
4.3.2 Inclure Conflict directed BackJumping with Reordering 55
4.4 Intégration de BTD 56
4.5 Implémentation 57
4.5.1 Pré-traitement sur les instances 57
4.5.2 Non déterminisme 57
4.6 Discussion 58

Partant de notre état de l'art, nous envisageons quelques voies pour apporter une pierre à l'édifice qu'est la programmation par contrainte. Nous souhaitons étudier en profondeur les bénéfices liés à l'exploitation de la structure d'une instance.

Les instances des problèmes réels issus de l'industrie ne sont pas des instances générées aléatoirement. Les instances de ces problèmes ont tendance à avoir une structure que l'on peut exploiter. Nous voudrions donc comparer finement les performances de BTD, qui exploite la structure des instances de façon statique, avec d'autres méthodes, qui l'exploitent de façon dynamique, en faisant des retours-arrières non chronologiques (comme CBJ, DBT ou DR) ou en collectant des informations sur la « difficulté » des variables (comme pour l'heuristique wdeg).

Pour effectuer cette comparaison nous réunissons différentes approches dans un même cadre. Nous présentons rapidement le cadre auquel nous voulons arriver en section 4.1. Nous partons du cadre de [LBH04], rappelé en section 4.2. Puis nous ajoutons à ce cadre de nouvelles stratégies de retour arrière exploitant dynamiquement la structure des CSP en section 4.3 et l'exploitation d'une décomposition arborescente en section 4.4. Enfin nous précisons les conditions d'implémentation de ce cadre en section 4.5.

4.1 Présentation du solveur générique

Le solveur générique que nous allons présenter nous permet de comparer dans un cadre uniforme les principales méthodes de l'état de l'art en résolution de CSP. Notre

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

but étant de comparer en particulier les méthodes utilisant une décomposition arborescente avec d'autres méthodes, il devient intéressant de proposer un cadre unifié.

Ce cadre générique est une extension de celui présenté par Lecoutre, Boussemart et Hemery dans [LBH04]. Le cadre original de [LBH04] comporte trois paramètres :

— un paramètre déterminant le type de retour-arrière, à choisir parmi CBT, CBJ et DBT ;
— un paramètre déterminant le niveau de propagation, à choisir parmi FC et MAC ;
— un paramètre déterminant l'heuristique de choix de variables, à choisir parmi ddeg et wdeg.

Nous décrivons en section 4.2 cet algorithme qui constitue la base de notre algorithme générique. Nous l'étendons en section 4.3 en y intégrant deux nouvelles stratégies de retour-arrière exploitant dynamiquement la structure d'une instance : CBJR et DR. Nous introduisons un quatrième paramètre en section 4.4. Ce paramètre est binaire et permet de spécifier si la recherche exploite une décomposition arborescente ou non de l'instance. Quand ce quatrième paramètre est instancié à $vrai$, si le type de retour-arrière est CBT, nous obtenons l'algorithme BTD, tandis que s'il est CBJ ou DR, nous obtenons de nouvelles stratégies de recherche. Nous n'avons pas étendu notre cadre au cas où le type de retour-arrière est CBJR ou DBT utilisés avec la décomposition arborescente.

Ainsi, notre cadre générique nous permettra de définir différentes configurations de solveurs. Chaque configuration est notée par un quadruplet (r, p, h, t) où :

— $r \in \{CBT, CBJ, DBT, CBJR, DR\}$ définit le mécanisme de retour-arrière ;
— $p \in \{FC, MAC\}$ définit le niveau de propagation des contraintes ;
— $h \in \{d, w\}$ définit l'heuristique de choix de variables ;
— et $t \in \{vrai, faux\}$ détermine si l'algorithme exploite une décomposition arborescente ou non.

Les configurations $(DBT, *, *, vrai)$ et $(CBJR, *, *, vrai)$ n'étant pas autorisées, nous obtenons $3 * 2 * 2 * 2 + 2 * 2 * 2 = 32$ configurations différentes qui seront évaluées et comparées au chapitre 6.

4.2 Rappel de l'algorithme de [LBH04]

Durant la recherche, certaines variables sont affectées et certains domaines sont réduits. À toute étape de la recherche, l'ensemble des variables est composé de variables affectées, appelées variables du passé et de variables non affectées, appelées variables futures. L'ensemble des variables futures est noté $varsF$ et étant donnée une contrainte c, $varsF(c)$ est équivalent à $portée(c) \cap varsF$. Le domaine initial d'une variable x est noté $D^{init}(x)$ et le domaine courant $D(x)$. Pour gérer les domaines, des explications d'élimination sont introduites. Une explication d'élimination $expl(x \neq v)$ pour une paire (x, v) composée d'une variable x et d'une valeur $v \in D^{init}(x)$ est un ensemble de variables tel qu'aucune solution contenant à la fois cet ensemble de variables et que x soit affecté à la valeur v n'existe. Dans la suite nous utiliserons l'équivalence suivante :

$$expl(x \neq v) = \emptyset \Leftrightarrow v \notin D(x)$$

Pour simplifier la suite, une affectation de la variable x à la valeur v sera représentée par x. En particulier, nous nous référerons aux explications d'éliminations comme si c'était un ensemble de variables plutôt qu'un ensemble d'affectations de variables.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Nous introduisons aussi une valeur spécifique notée $FINAL$, qui est équivalente à l'ensemble vide ($\forall E, E \cup FINAL = E, \forall x, x \notin FINAL$) mais est différente de l'ensemble vide ($FINAL \neq \emptyset$). Cette valeur sera utilisée pour représenter toute explication d'élimination correspondant à une valeur supprimée définitivement. Nous avons besoin de cette valeur car l'ensemble vide pour une explication d'élimination pour une variable x et une valeur v indique que la valeur éliminée ne fait pas partie du domaine initial de x, $D^{init}(x)$.

La recherche est commencée en appelant la fonction *rechercheItérative* (voir algorithme 10) avec un CSP à résoudre. Après avoir choisi les méthodes de retour arrière et de propagation, une boucle principale essaie différentes affectations pour les variables. Quand une affectation ($x \leftarrow v$) doit être effectuée (voir algorithme 11), toutes les valeurs présentes dans D(x) sauf v sont enlevées en donnant l'affectation de x comme explication. Après chaque affectation la fonction *vérifierCohérenceAprèsAffectation* (décrite plus loin) est appelée pour déterminer si un nogood (ensemble d'affectations incompatibles) peut être identifié. Dans un tel cas, la fonction *gérerContradiction* est appelée. Enfin, soit une solution est trouvée, soit l'incohérence est détectée (la recherche peut être arrêtée par l'algorithme 15 qui est appelé par *gérerContradiction*).

Algorithme 10 : rechercheItérative (X, D, C)

Entrées : Un CSP $P = (X, D, C)$
Sorties : $vrai$ si P est cohérent, $faux$ si P est incohérent

```
1  choisir retourArriere dans {CBT, CBJ, DBT}
2  choisir propagation dans {FC, AC}
3  varsF ← X
4  tant que varsF ≠ ∅ faire
5      x ←choixVariable(varsF)
6      v ←choixValeur(D(x))
7      affecter(x,v)
8      conflits ←vérifierCohérenceAprèsAffectation(x)
9      si conflits ≠ ∅ alors
10         si ¬ gérerContradiction(conflits) alors
11             retourner faux
12 retourner vrai
```

Algorithme 11 : affecter (x, v)

Entrées : une variable x et une valeur v
Sorties : rien, v est affectée à x

```
1  varsF ← varsF \ {x}
2  pour chaque v' ∈D(x), v' ≠ v faire
3      expl(x ≠ v') ← {x}
```

Tant qu'une contradiction doit être gérée (algorithme 12), l'affectation de la variable coupable la plus récente dans l'ensemble $conflits$ est choisie pour être défaite (ainsi que toutes les affectations suivantes pour CBJ). Notons que pour CBT, la variable coupable la plus récente est forcément la dernière affectée. Ensuite la valeur affectée peut être enlevée (algorithme 14), en donnant une explication, puisque la portion de l'es-

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

pace de recherche correspondante vient d'être explorée. Si l'ensemble de conflits est vide, alors la valeur peut être enlevée définitivement. Sinon, l'ensemble de conflits correspond aux explications de la valeur, sauf pour CBT où seule la dernière variable affectée est considérée coupable. Après avoir retiré la valeur, si le domaine de x est vide nous le gérons (avec la fonction *gérerDomaineVide*). Sinon, la cohérence est vérifiée avec la fonction *vérifierCohérenceAprèsReduction* (décrite plus loin).

Algorithme 12 : gérerContradiction $(conflits)$

```
   Entrées : un ensemble de variables conflits
   Sorties : faux si le problème est prouvé incohérent, vrai sinon
1  tant que conflits ≠ ∅ faire
2      soit (x, v) la dernière affectation de variable avec x ∈ conflits
3      enlever x de conflits
4      si retourArriere = CBJ alors
5          tant que la dernière variable affectée x' est telle que x' ≠ x faire
6              défaireAffectation(x')
7      défaireAffectation(x)
8      enleverValeur(x, v, conflits)
9      si D(x) = ∅ alors
10         conflits ←gérerDomaineVide(x)
11         si conflits = ∅ alors
12             retourner faux
13     sinon
14         conflits ← vérifierCohérenceAprèsRéduction(x)
15 retourner vrai
```

Quand une affectation de variable est défaite (algorithme 13), il est nécessaire d'enlever toutes les explications contenant cette variable. Pour des valeurs apparaissant dans le domaine de variables affectées, nous pouvons produire immédiatement de nouvelles explications. Pour des valeurs apparaissant dans le domaine de variables non affectées, nous vérifions qu'il n'existe pas d'autres explications. Cependant, cette recherche d'explications est retardée jusqu'à l'appel de la fonction *vérifierCohérenceAprèsAffectation*.

Algorithme 13 : défaireAffectation (x)

```
   Entrées : une variable x
   Sorties : rien, x est désaffectée
1  varsF ← {x} ∪ varsF
2  pour chaque (x', v'), x' ∈ X ∧ v' ∈D^init(x')∧x ∈ expl(x' ≠ v') faire
3      si x' ∈ varsF alors
4          expl(x' ≠ v') ← ∅
5      sinon
6          expl(x' ≠ v') ← {x'}
```

Enfin, quand un domaine vide est détecté, la fonction *gérerDomaineVide* (algorithme 15) est appelée. Elle collecte les explications pour les valeurs enlevées, sauf pour CBT pour

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Algorithme 14 : enleverValeur $(x, v, conflits)$

Entrées : une variable x, une valeur v et un ensemble de variables $conflits$
Sorties : rien, v est enlevée du domaine de x

```
1 si conflits = ∅ alors
2     expl(x ≠ v) ← FINAL
3 sinon si retourArriere ≠ CBT alors
4     expl(x ≠ v) ← conflits
5 sinon
6     expl(x ≠ v) ← {x'} où x' est la dernière variable affectée
```

qui la dernière variable affectée est l'unique coupable. S'il n'y a pas de variable coupable, la recherche est terminée et le CSP est prouvé incohérent.

Algorithme 15 : gérerDomaineVide (x)

Entrées : une variable x
Sorties : un ensemble de variables causant l'échec en x

```
1 conflits ← ∅
2 si retourArriere ≠ CBT alors
3     pour chaque v ∈D^init(x) faire
4         conflits ← conflits ∪ expl(x, v)
5 sinon si (X \ varsF) ≠ ∅ alors
6     conflits ← {x'} où x' est la dernière variable affectée
7 retourner conflits
```

Après chaque affectation de variable, nous devons vérifier la cohérence du problème (algorithme 16). Un ensemble de propagation Q contient les arcs (paires de contraintes et variables) devant être révisés. L'objectif de la révision d'un arc (c, x) est de retirer toutes les valeurs incohérentes de D(x) vis à vis de la contrainte c. L'ensemble Q est initialisé avec les contraintes contenant la dernière variable affectée et une unique autre variable. Dès que Q est initialisé, la propagation de contraintes est lancée avec la fonction *filtrer* (algorithme 18). L'article [LBH04] ne précise pas la version de AC utilisée ; nous utilisons AC2001 avec CBT et BTD, et AC3 avec toutes les autres stratégies de retour arrière (puisqu'elle enregistrent déjà les informations nécessaires pour AC3).

Quand, après une désaffectation, une valeur est retirée du domaine d'une variable (algorithme 12) et que le domaine n'est pas vide nous devons vérifier la cohérence du problème (algorithme 17). En effet, retirer cette valeur doit être pris en compte et de plus, certaines valeurs remises dans les domaines peuvent être présentes dans d'autres explications. Quand FC est combiné avec DBT, nous devons vérifier qu'aucune des valeurs (remises dans les domaines) n'apparaisse dans des explications (comme DBT ne désaffecte pas l'ordre chronologique d'affectation). En effet, le filtrage enregistre de nouvelles raisons de retirer la valeur ou des explications permettant de filtrer plusieurs fois la même valeur et dès qu'une explication n'est plus valide nous en cherchons une nouvelle. Dans cet algorithme 17, quand AC et CBT sont combinés, il faut maintenir l'arc-cohérence sur la variable réduite tandis qu'avec AC et DBT il faut vérifier qu'aucune variable n'apparaisse dans des explications, et initialiser complètement

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Algorithme 16 : vérifierCohérenceAprèsAffectation (x)

Entrées : une variable x
Sorties : $\emptyset$ s'il n'y a pas d'incohérence détectée, un ensemble de variables ayant participé à vider le domaine d'une variable sinon

```
1 suivant propagation faire
2     cas où FC
3         Q ← {(c, x') | c ∈ C ∧ x ∈ portée(c)∧{x'} = varsF(c)}
4     cas où AC
5         Q ← {(c, x') | c ∈ C ∧ x ∈ portée(c)∧x' ∈ varsF(c)}
6 retourner filtrer(Q)
```

l'ensemble Q des arcs à vérifier. L'ensemble Q est également complètement initialisé quand AC et CBJ sont utilisés puisque plusieurs désaffectations peuvent se suivre et l'arc-cohérence n'est pas nécessairement garantie lorsqu'un domaine se vide.

Algorithme 17 : vérifierCohérenceAprèsReduction (x)

Entrées : une variable x
Sorties : $\emptyset$ s'il n'y a pas d'incohérence détectée, un ensemble de variables ayant participé à vider le domaine d'une variable sinon

```
1  suivant propagation faire
2      cas où FC
3          si retourArriere ≠ DBT alors
4              Q ← ∅
5          sinon
6              Q ← {(c, x') | c ∈ C ∧ {x'} = varsF(c)}
7      cas où AC
8          si retourArriere = CBT alors
9              Q ← {(c, x') | c ∈ C ∧ x ∈ portée(c)∧x' ∈ varsF(c) ∧ x' ≠ x}
10         sinon
11             Q ← {(c, x') | c ∈ C ∧ x' ∈ varsF(c)}
12 retourner filtrer(Q)
```

Le filtrage est effectué par la fonction *filtrer* (voir algorithme 18). Chaque arc est révisé et dès qu'une valeur est supprimée, il faut vérifier qu'aucun domaine ne s'est vidé. Si tel est le cas il faut appeler la fonction *gérerDomaineVide*. Sinon, l'ensemble Q est mis à jour si AC est utilisé.

Une révision est effectuée par la fonction *réviser* (algorithme 19) en enlevant les valeurs de D(x) devenues incohérentes vis à vis de c. Chaque valeur supprimée reçoit une explication avec la fonction *donneExpls* (algorithme 20). Cette fonction est appelée pour fournir une explication. Pour CBT, la dernière variable affectée est l'unique coupable. Pour CBJ et DBT, les explications des valeurs support de (x, v) dans c sont réunies. Une valeur sans explication est définitivement retirée.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Algorithme 18 : filtrer (Q)

Entrées : un ensemble de paires de contraintes et variables Q
Sorties : $\emptyset$ s'il n'y a pas d'incohérence détectée, un ensemble de variables ayant participé à vider le domaine d'une variable sinon

```
1 tant que Q ≠ ∅ faire
2     choisir (c, x) ∈ Q
3     si réviser(c, x) alors
4         si D(x) = ∅ alors
5             poids[c] + +                                   // pour wdeg
6             retourner gérerDomaineVide(x)
7         sinon si propagation = AC alors
8             Q ← Q ∪ {(c', x') | x ∈portée(c')∧x' ∈portée(c')∧x ≠ x' ∧ c ≠ c'}
9 retourner ∅
```

Algorithme 19 : réviser (c, x)

Entrées : une contrainte c et une variable x
Sorties : $vrai$ si c retire une valeur du domaine de x, $faux$ sinon

```
1 nbElements ←|D(x)|
2 pour chaque x ∈D(x) faire
3     si estCohérent(c, x, v)= faux alors
4         expl(x ≠ v) ←donneExpls(c, x, v)
5 retourner nbElements ≠|D(x)|
```

Algorithme 20 : donneExpls (c, x, v)

Entrées : une contrainte c, une variable x et une valeur v
Sorties : un ensemble de variables

```
1 conflits ← ∅
2 si retourArriere ≠ CBT alors
3     pour chaque (x', v') | x' ∈portée(c)∧x' ≠ x ∧ v' ∈ (D^init(x')\D(x')) faire
4         si il y a un support pour à la fois (x, v) et (x', v') dans c alors
5             conflits ← conflits ∪ expl(x' ≠ v')
6 sinon si (X \ varsF) ≠ ∅ alors
7     conflits ← {x'} où x' est la dernière variable affectée
8 si conflits = ∅ alors
9     conflits ← FINAL
10 retourner conflits
```

4.3 Extensions de l'algorithme à de nouvelles stratégies dynamiques

Pour commencer à enrichir le cadre de Lecoutre, Hémery et Boussemart nous y ajoutons les stratégies de retour arrière DR et CBJR.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

4.3.1 Inclure Decision Repair

La méthode Decision Repair repose sur les mêmes techniques que CBJ et DBT déjà capturées par le cadre générique. La différence fondamentale réside dans le fait de ne plus nécessairement désaffecter la dernière variable en cause dans un échec (et les variables intermédiaires pour CBJ), mais de pouvoir choisir librement une parmi toutes ces variables. Une conséquence de cette liberté est que cette méthode n'assure plus la propriété de complétude. À l'image des méthodes incomplètes, si elle parvient à construire une solution en affectant de manière cohérente toutes les variables du problème, alors elle pourra retourner $vrai$ pour traduire la cohérence du problème. Mais, contrairement à la grande majorité des méthodes incomplètes, elle est capable de prouver l'incohérence dans un cas particulier : si elle rencontre un échec et que l'ensemble des variables en cause est vide. Il se trouve que le cadre est construit de telle manière que ce cas précis est l'unique possibilité pour les méthodes capturées de prouver l'incohérence d'un problème. Il devient dès lors assez aisé d'étendre le cadre afin de capturer DR. Il suffit de modifier la fonction *gérerContradiction* (comparer les algorithmes 12 et 21) afin de choisir une variable parmi celles en cause, pas nécessairement la dernière et sans avoir à désaffecter les variables intermédiaires.

Algorithme 21 : gérerContradictionDR $(conflits)$

```
   Entrées : un ensemble de variables conflits
   Sorties : faux si le CSP est prouvé incohérent, vrai sinon
1  tant que conflits ≠ ∅ faire
2      si retourArriere = DR alors
3          choisir (x, v) avec x ∈ conflits
4      sinon
5          soit (x, v) la dernière affectation de variable avec x ∈ conflits
6      enlever x de conflits
7      si retourArriere = CBJ alors
8          tant que la dernière variable affectée x' est telle que x' ≠ x faire
9              défaireAffectation(x')
10     défaireAffectation(x)
11     enleverValeur(x, v, conflits)
12     si D(x) = ∅ alors
13         conflits ←gérerDomaineVide(x)
14         si conflits = ∅ alors
15             retourner faux
16     sinon
17         conflits ← vérifierCohérenceAprèsRéduction(x)
18 retourner vrai
```

De plus, dans la fonction *vérifierCohérenceAprèsReduction* (voir algorithme 17) la condition de la ligne 3 devient : si $retourArriere \neq DBT \wedge retourArriere \neq DR$.

Il faut noter que contrairement aux autres méthodes (complètes) capturées par le cadre générique, DR ne garantit pas la propriété de terminaison. En effet, son espace de recherche n'est pas structuré en arbre, mais en graphe à l'image de DBT. Là où le retour à la dernière variable en cause dans l'échec assure la terminaison de DBT, le fait

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

de pouvoir désaffecter n'importe quelle variable en cause peut mener DR à rester bloqué dans un cycle avec par exemple deux variables alternativement parmi les causes de leurs échecs respectifs. L'heuristique MinDestroy [PV04], en choisissant de manière aléatoire parmi toutes les variables en causes celles minimisant le nombre d'explications de suppression de valeur les concernant, réduit considérablement les possibilités de rester bloqué dans un cyle. Par ailleurs, cette heuristique permet de conserver un maximum d'explications et de remettre dans les domaines un minimum de valeurs afin de faciliter une éventuelle preuve d'incohérence.

L'heuristique MinDestroy renvoie donc : $\underset{x \in conflits}{\operatorname{argmin}} \mid \{expl \in CS, x \in expl\} \mid$, où CS est l'ensemble contenant toutes les explications.

4.3.2 Inclure Conflict directed BackJumping with Reordering

Pour inclure CBJR dans ce solveur générique, il faut pouvoir manipuler l'affectation avec une fonction *remonter* qui permet de monter une variable d'un niveau dans l'affectation. Nous modifions donc la boucle principale *rechercheItérative* (comparer les algorithmes 10 et 22). Le retour arrière CBJR remonte la variable nouvellement affectée tant que celle-ci n'est pas en conflit avec la variable avant elle dans l'affectation et que l'heuristique de choix de variable donne un score plus important à la variable à remonter. Le score de l'heuristique de choix de variable est donné par la fonction $score(x)$ ou x est une variable. La fonction $expls(x)$ donne l'ensemble des variables impliquées dans la suppression d'au moins une valeur du domaine de x.

Algorithme 22 : rechercheItérativeCBJR (X, D, C)

Entrées : Un CSP $P = (X, D, C)$
Sorties : $vrai$ si P est cohérent, $faux$ si P est incohérent

```
1  choisir retourArriere dans {CBT, CBJ, CBJR, DBT, DR}
2  choisir propagation dans {FC, AC}
3  varsF ← X
4  tant que varsF ≠ ∅ faire
5      x ←choixVariable(varsF)
6      v ←choixValeur(D(x))
7      affecter(x,v)
8      conflits ←vérifierCohérenceAprèsAffectation(x)
9      si conflits ≠ ∅ alors
10         si ¬ gérerContradiction(conflits) alors
11             retourner faux
12     sinon si conflits = ∅ ∧ retourArriere = CBJR alors
13         Soit y la variable au-dessus de x dans l'affectation courante
14         tant que y ∉ expls(x) ∧ score(x) > score(y) faire
15             remonter(x)
16             Soit y la variable au-dessus de x dans l'affectation courante
17 retourner vrai
```

Pour le reste, CBJR se comporte comme CBJ, il convient donc de mettre une condition équivalente à CBJ pour CBJR à l'algorithme 21 ligne 7 : si $retourArriere = CBJ \vee retourArriere = CBJR$.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

4.4 Intégration de BTD

L'algorithme 6 présentant BTD au chapitre 2 est générique dans le sens où il peut utiliser différentes approches de résolution pour calculer une affectation cohérente pour les variables d'un cluster donné : cela est fait par la méthode $next()$, qui retourne à chaque appel une affectation cohérente différente, ou $null$ s'il n'existe plus d'affectation cohérente.

Nous proposons d'étendre le cadre générique de [LBH04] rappelé en section 4.2 en ajoutant un quatrième paramètre t tel que lorsque $t = vrai$, l'algorithme utilise une décomposition arborescente pour guider la résolution. Dans ce cas, il s'agit tout d'abord de calculer une décomposition arborescente du graphe des contraintes et de choisir un cluster racine, selon le principe rappelé au chapitre 2 (section 2.2.4). Ensuite, l'algorithme 6 est exécuté en lui passant en paramètres le CSP, sa décomposition arborescente et le cluster racine.

Si $retourArriere = CBT$, alors l'algorithme 10 peut être utilisé pour calculer les différentes affectations cohérentes d'un cluster, en limitant l'ensemble $varsF$ aux variables du cluster (ligne 3 de l'algorithme 10). Au lieu de retourner $vrai$ (ligne 11), l'algorithme 10 suspend son exécution et retourne les valeurs affectées à $varsF$. Lors de l'appel suivant à $next()$, l'algorithme 10 reprend son exécution en défaisant la dernière affectation pour rechercher une nouvelle solution.

Si $retourArriere \in \{CBJ, DR\}$, nous devons également modifier l'algorithme 21 qui est appelé lorsqu'une contradiction a été détectée. Dans ce cas, l'algorithme 21 est remplacé par l'algorithme suivant, où E_r désigne le cluster racine pour l'appel courant de BTD, et E_s désigne l'ensemble des variables du séparateur de E_r (si l'appel courant de BTD est le premier, de sorte que E_r est la racine de la décomposition initiale, alors $E_s = \emptyset$).

Si tous les conflits portent sur des variables du séparateur E_s (ligne 2), alors l'exécution de l'algorithme 23 se termine en retournant $faux$, de sorte que la fonction $next()$ de l'algorithme 6 retourne $null$. Dans ce cas, l'algorithme 6 retourne un échec pour E_r, on revient dans l'exécution de l'algorithme 6 sur le parent de E_r, et on enregistre un nogood pour les variables du séparateur E_s.

Si certains conflits portent sur des variables qui n'appartiennent pas au séparateur, alors on retrouve le principe de l'algorithme de gestion des contradictions précédent (algorithme 21), à la différence que pour DR, nous limitons le choix de la variable à désaffecter à l'ensemble des variables en conflit qui n'appartiennent pas au séparateur E_s (ligne 6).

Notons que lorsque la fonction $next()$ est rappelée par l'algorithme 6 pour chercher une nouvelle affectation cohérente (ligne 2), il est nécessaire de garantir que la nouvelle affectation retournée par $next()$ est différente des affectations retournées par les appels précédents. Si cela est évident quand $retourArriere \neq DR$, ce n'est pas nécessairement le cas quand $retourArriere = DR$. Remarquons maintenant que la fonction $next()$ (ligne 2 de l'algorithme 6) ne peut être appelée plusieurs fois que si un appel récursif à BTD (ligne 8) sur un fils de r retourne échec (si tous les fils retournent succès, alors l'algorithme 6 se termine sur un succès, ligne 13). Soit i le fils de r pour lequel l'appel à BTD de la ligne 8 a retourné échec. Pour empêcher DR de retourner la même affectation cohérente pour les variables de E_r lors du prochain appel à la fonction $next()$ (ligne 2), nous ajoutons la contrainte $\neg A_i$ à l'ensemble C des contraintes du modèle.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Algorithme 23 : gérerContradictionBTD ($conflits$)

Entrées : Un ensemble de variables $conflits$
Sorties : rien ; si le problème est prouvé incohérent l'exécution est terminée, sinon un ensemble de variable causant l'échec courant est calculé

```
1  tant que conflits ≠ ∅ faire
2      si toutes les variables de conflits appartiennent à E_s alors
3          terminer l'exécution de l'algorithme 10 en retournant faux
4      sinon
5          si retourArriere = DR alors
6              choisir une variable x ∈ conflits ∩ E_r \ E_s
7          sinon
8              soit (x, v) la dernière affectation de variable avec x ∈ conflits
9          enlever x de conflits
10         si retourArriere = CBJ alors
11             tant que la dernière variable affectée x' est telle que x' ≠ x faire
12                 défaireAffectation(x')
13         défaireAffectation(x)
14         enleverValeur(x, v, conflits)
15         si D(x) = ∅ alors
16             conflits ←gérerDomaineVide(x)
17         sinon
18             conflits ←vérifierCohérenceAprèsRéduction(x)
```

4.5 Implémentation

Notre algorithme générique a été implémenté en C. Nous décrivons ici le pré-traitement effectué sur les instances, pour toutes les configurations du solveur (section 4.5.1), ainsi que les règles utilisées pour rendre notre algorithme non déterministe.

4.5.1 Pré-traitement sur les instances

Pour toutes les configurations, nous commençons par décomposer le graphe des contraintes en ses composantes connexes pour obtenir des sous problèmes indépendants, qui sont résolus consécutivement. Enfin, chaque sous problème est rendu arc-cohérent avec AC3 avant d'entamer la résolution, même pour les configurations utilisant FC. Ceci est fait car ces pré-traitements sont peu coûteux en temps de calcul mais peuvent être très efficace.

4.5.2 Non déterminisme

Notre algorithme générique peut être aisément rendu déterministe en imposant des règles arbitraires de choix (par exemple, l'ordre alphabétique pour choisir les variables en cas d'ex-æquo, l'ordre alphabétique pour choisir les valeurs, ...).

Plutôt que d'imposer des règles arbitraires, nous avons choisi de rendre notre solveur non déterministe : lors du choix d'une variable, les égalités sont départagées aléatoirement ; de plus nous n'utilisons pas d'heuristique de choix de valeur et les valeurs

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

sont aussi choisies aléatoirement.

Cela est implémenté en fixant un ordre arbitraire sur les variables (respectivement les valeurs) et en parcourant les variables (respectivement les valeurs) selon cet ordre, mais en choisissant de façon aléatoire la première variable (respectivement valeur) à partir de laquelle le parcours est commencé.

Il sera intéressant d'étudier l'impact du non déterminisme à l'aide de multiples exécutions de mêmes configurations sur les mêmes instances. Des tests statistiques nous aiderons à analyser les résultats.

4.6 Discussion

Nous avons maintenant un cadre générique nous permettant une comparaison aisée d'anciennes et de nouvelles configurations.

En plus de la comparaison de différents types de retours arrière intelligents, notre cadre générique permet de comparer différentes exploitations de la structure de CSP. Il est intéressant de noter qu'il existe différentes notions de structure pour un CSP : nous avons abordé l'exploitation statique de la structure grace à la décomposition arborescente, mais il existe aussi d'autres types d'exploitation de la structure. Nous énumérons ici les éléments du solveur générique qui ont pour objectif d'exploiter différemment la structure d'un CSP.

Retours arrière intelligents

Comme vu à la section 2.2.3 les retours arrière intelligents permettent de sauvegarder de l'information plutôt que de la détruire à chaque échec. Cela induit une découverte dynamique de la structure d'un CSP. C'est flagrant dans le cas de DBT et donc aussi de DR. Pour CBJ l'idée est de rester localement sur la cause d'un échec sans se préoccuper du reste, de le résoudre avant de poursuivre la recherche : c'est donc une détection dynamique de conflit à résoudre, il y a donc une découverte de la structure sous-jacente.

Heuristique dynamique pour le choix de variable

Comme présenté en section 2.2.5, wdeg est une heuristique de choix de variable qui, en pondérant les variables par le nombre de fois où elles déclenchent un échec, détecte les endroits épineux d'un problème et donc sa structure.

Dans une moindre mesure, ddeg exploite également la structure d'une instance en favorisant les variables liées par une contrainte au plus grand nombre de variables non affectées.

Utiliser la décomposition arborescente

Nous avons inclus BTD dans notre solveur générique afin de comparer l'exploitation statique (ici avec une décomposition arborescente) de l'exploitation dynamique de la structure (avec wdeg ou les retours arrière intelligents par exemple).

La méthode BTD a comme avantage de restreindre la recherche par cluster de la décomposition arborescente. Cela nous garantit de ne pas éparpiller l'effort de recherche. De plus la complexité en temps de cet algorithme est limitée non plus par le nombre de variables total du problème mais par la taille maximale d'un cluster. Nous pouvons

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

donc nous attendre à ce qu'un CSP avec une bonne décomposition arborescente (c'est-à-dire une décomposition avec une largeur faible, voir 2.2.4) soit résolu efficacement par BTD.

Nous avons donc présenté notre cadre générique permettant la comparaison directe de beaucoup de configurations. Nous avons discuté des points qui nous semblent les plus intéressants à analyser lors de nos expérimentations. Enfin nous avons décrit succinctement l'implémentation que nous avons réalisée de ce cadre générique et de ses spécificités, en particulier la prise en compte du non déterminisme implicite.

Il nous reste à présenter l'ensemble d'instances sur lequel nous allons réaliser nos expérimentations.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Chapitre 5

Instances considérées pour l'évaluation

Sommaire

5.1 Instances au format XCSP 61
5.1.1 Élimination de certaines instances 62
5.1.2 Les différentes classes d'instances 62
5.2 Instances structurées générées aléatoirement 64
5.2.1 Méthode de génération 64
5.2.2 Paramètres retenus pour générer un ensemble d'instances . . . 66
5.3 Discussion 66

Dans ce chapitre nous allons présenter les différents groupes d'instances que nous avons récupérées et leur provenance, afin de tester le cadre générique présenté au chapitre précédent.

Les instances récupérées sont, pour une première partie, de la compétition CSP'08 (voir section 5.1) et pour la seconde partie générées aléatoirement avec une structure statique exploitable via une décomposition arborescente (voir section 5.2).

5.1 Instances au format XCSP

La première partie des instances que nous avons considérées vient de la compétition CSP'08 [1] que l'on peut trouver, avec de nouvelles instances ajoutées depuis la fin de la compétition, sur le site http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html. Cet ensemble d'instances contient des instances académiques (puzzles, coloriages de graphes, ...) aussi bien que des instances issues de l'industrie (comme les instances du CELAR qui a proposé les premières instances RLFAP, des instances du problème d'allocation de fréquences).

Nous allons présenter la méthode utilisée pour constituer notre ensemble d'instances à partir de cette ressource, puis nous présenterons les différentes classes de cet ensemble d'instances.

1. http://www.cril.univ-artois.fr/CPAI08/

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Classe	#I	#Variables			#Valeurs			#Contraintes			Dureté des contraintes		
		min	moy	max	min	moy	max	min	moy	max	min	moy	max
ACAD	75	10	116	500	2	146	2187	45	691	4950	0.001	0.692	0.998
PATT	238	16	263	1916	3	66	378	48	4492	65390	0.002	0.795	0.996
QRND	80	50	220	315	7	11	20	451	2968	4388	0.122	0.578	0.823
RAND	206	23	37	59	8	36	180	84	282	753	0.095	0.613	0.984
REAL	193	200	628	1000	2	152	802	1235	6394	17447	0.001	0.519	0.999
total	792	10	274,9	1916	2	81,2	2187	45	3346,6	65390	0,001	0,649	0,999

Tableau 5.1 – *Classes de l'ensemble d'instances. Pour chaque classe, le tableau donne son nom, son nombre d'instances (#I), son nombre de variables, la taille de ses domaines, son nombre de contraintes et la dureté des contraintes (ratio des tuples interdits sur le nombre de tuples possibles) : le minimum, en moyenne et le maximum.*

5.1.1 Élimination de certaines instances

Dans cet ensemble d'instances nous avons sélectionné un sous-ensemble comme suit :

— nous choisissons de ne considérer que les instances binaires ;
— si une classe d'instances contient plus de 10 instances, alors nous ne gardons que les dix premières, ceci afin d'éviter une sur-représentation d'une classe d'instances ;
— si une instance n'est résolue par aucune configuration de notre cadre générique dans une limite de 30 minutes sur 15 essais, nous l'enlevons de notre ensemble de problème.

Nous obtenons ainsi 792 instances. Nous pouvons maintenant présenter les caractéristiques des classes de ces instances.

5.1.2 Les différentes classes d'instances

L'ensemble d'instances XCSP est divisé en plusieurs classes, chaque classe d'instances contenant plusieurs types de problèmes. Les cinq grandes classes de problèmes sont :

— ACAD, contenant des instances de problèmes « académiques » générés sans recours à l'aléatoire, tels que le problème des tours de Hanoï, ou encore des problèmes d'assemblage de dominos ;
— PATT, contenant des instances suivant des motifs réguliers avec l'aide de génération aléatoire, cette classe contient beaucoup d'instances de problèmes de coloriage de graphes ;
— QRND, contenant des instances aléatoires avec une structure – structure différente de celle présentée en section 5.2 ;
— RAND, contenant des instances générées complètement aléatoirement ;
— enfin REAL, contenant des instances de problèmes issus de l'industrie, comme par exemple des problèmes d'allocation de fréquences avec les instances CELAR.

Le tableau 5.1 présente les caractéristiques des cinq classes d'instances. Nous constatons la diversité des instances de cet ensemble de problèmes. La plus petite instance a 10 variables, la plus grande en a 1916. Il y a une instance qui est définie avec plus de 65000 contraintes. Notons que la classe REAL contient des instances de taille conséquente : les instances de cette classe ont 628 variables en moyenne, et 152 valeurs en moyenne.

Sur ces instances, la largeur arborescente calculée est inférieure ou égale à $\frac{|X|}{2}$ sur

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Nom (abrév)	S	#X	#C	MD	MC	MS
Instances PATT						
anna-8 (an8)	I	138	493	8	13	12
anna-9 (an9)	I	139	494	9	13	12
david-8 (da8)	I	87	406	8	14	12
david-9 (da9)	I	87	406	9	14	12
david-10 (da10)	I	87	406	10	14	12
homer-8 (ho8)	I	561	1628	8	32	30
huck-8 (hu8)	I	74	301	8	11	6
huck-9 (hu9)	I	74	301	9	11	6
huck-10 (hu10)	I	74	301	10	11	6
jean7 (j7)	I	80	254	7	10	8
jean8 (j8)	I	80	254	8	10	8
jean9 (j9)	I	80	254	9	10	8
mug88-1-3 (m1)	I	88	146	3	4	2
mug88-25-3 (m2)	I	88	146	3	4	2
mug100-1-3 (m3)	I	100	166	3	4	2
mug100-25-3 (m4)	I	100	166	3	4	2
Instances REAL						
graph2-f24 (gm2)	C	400	2245	22	157	29
graph8-f11 (gm8)	I	680	3757	33	330	30
graph9-f10 (gm9)	I	916	5246	34	387	29
graph12-w1 (gm12)	I	680	1148	44	49	30
graph13-w1 (gm13)	I	916	1479	44	89	28
graph1 (g1)	C	200	1134	44	72	28
graph2 (g2)	C	400	2245	44	157	29
graph3 (g3)	C	200	1134	44	69	27
graph8 (g8)	C	680	3757	44	330	30
graph9 (g9)	C	916	5246	44	387	29
graph14 (g14)	C	916	4638	44	412	30
scen1-f8 (sm1)	C	916	5548	36	33	28
scen1-f9 (sm2)	I	916	5548	35	33	28
scen2-f25 (sm3)	I	200	1235	21	21	17
scen6-w1-f2 (sm4)	I	200	319	42	8	7
scen6-w1 (sm5)	C	200	319	44	8	7
scen6-w2 (sm6)	I	200	648	44	14	12
scen7-w1-f4 (sm7)	C	400	660	40	8	7
scen7-w1-f5 (sm8)	I	400	660	39	8	7
scen9-w1-f3 (sm9)	I	680	1138	41	8	7
scen10-w1-f3 (sm10)	I	680	1138	41	8	7
scen1 (s1)	C	916	5548	44	33	28
scen3 (s3)	C	400	2760	44	34	29
scen4 (s4)	C	680	3967	44	33	28
scen5 (s5)	C	400	2598	44	33	28
scen11 (s11)	C	680	4103	44	33	28
scen06-sub0 (su6)	I	32	223	44	16	12
scen07-sub4 (su7)	I	44	499	44	21	19
Instance ACAD						
hanoi-7_ext (ha7)	I	126	125	2187	7	1

Tableau 5.2 – *Instances XCSP dont la décomposition arborescente a une largeur inférieure ou égale à $\#X/2$. Pour chaque instance nous donnons, son nom, son abréviation, si elle est (S) cohérente (C) ou incohérente (I), son nombre de variables (#X), son nombre de contraintes (#C), la taille maximum des domaines (MD), la taille maximum d'un cluster dans la décomposition (MC) et la taille de la plus grande intersection entre deux clusters de la décomposition (MS).*

seulement 75 instances. Pour le reste des instances, où la largeur arborescente est strictement supérieure à $\frac{|X|}{2}$, l'utilité de méthodes comme BTD est sérieusement remise en

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

cause.

Cela nous montre que très peu d'instances de cet ensemble possèdent une structure pouvant être exploitée par une décomposition arborescente ayant une largeur arborescente raisonnable. En particulier, aucune instance de RAND (instances générées aléatoirement) n'a de structure (exploitable par une décomposition arborescente). Sans surprise, la classe contenant le plus d'instances structurées est REAL contenant les instances issues de l'industrie avec 61 instances. Les classes PATT et ACAD contiennent 33 et 5 instances respectivement.

Parmi les 99 instances structurées, il y a 24 instances trop difficiles (aucune configuration ne les résout en 30 minutes) et il y a 30 instances trop faciles (résolues en moins d'une seconde par toutes les configurations de notre cadre générique). Les 45 instances restantes sont présentées dans le tableau 5.2. L'instance hanoi* est issue d'un problème de satisfaction ; les instances scen* et graph* sont des problèmes d'allocation de fréquences issus de l'industrie ; et les instances de PATT sont des instances de coloration de graphes transformées en problèmes de satisfaction en incrémentant, pour chaque instance, le nombre de couleurs disponibles jusqu'à ce qu'une solution existe.

Il y a des instances dont la solution est trouvée en prenant la première valeur de chaque domaine. Pour une telle instance, il devient absolument nécessaire de choisir aléatoirement l'ordre d'affectations des valeurs, faute de quoi la résolution est quasi instantanée et ne nous apprend rien sur le comportement de notre configuration.

Certaines instances sont résolues immédiatement en appliquant la cohérence d'arc. C'est le cas par exemple du problème des tours de Hanoï avec un paramètre compris entre 3 et 6.

5.2 Instances structurées générées aléatoirement

Pour avoir des instances dont nous pouvons contrôler la structure, nous utilisons des instances générées aléatoirement avec une structure. Ces instances sont générées avec une structure de même type que les instances RLFAP (voir la figure 5.1 pour une instance du CELAR), qui sont issues de problèmes d'allocation de fréquences venant de l'industrie. Ces problèmes pourront départager les méthodes capables d'exploiter une structure – que ce soit de façon statique ou de façon dynamique – pour résoudre une instance. La stratégie de retour arrière BTD devrait être performante sur ces instances. Ces instances seront appelées STRUCT dans la suite.

5.2.1 Méthode de génération

Nous utilisons ici la même méthode que dans [JNT06] pour générer des instances dont nous contrôlons la largeur d'arbre et la difficulté. Une classe d'instances structurées est définie par les paramètres (n, d, w, t, s, ns, p) tels que :

- n est le nombre de variables de l'instance
- d est le nombre de valeurs dans chaque domaine d'une variable (les domaines sont uniformes)
- w est la borne maximale sur la largeur d'arbre du graphe des contraintes
- t permet de contrôler la dureté des contraintes et est le pourcentage de tuples interdits dans une contrainte
- s est la borne maximale sur la taille des séparateurs de la décomposition arborescente du graphe des contraintes

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

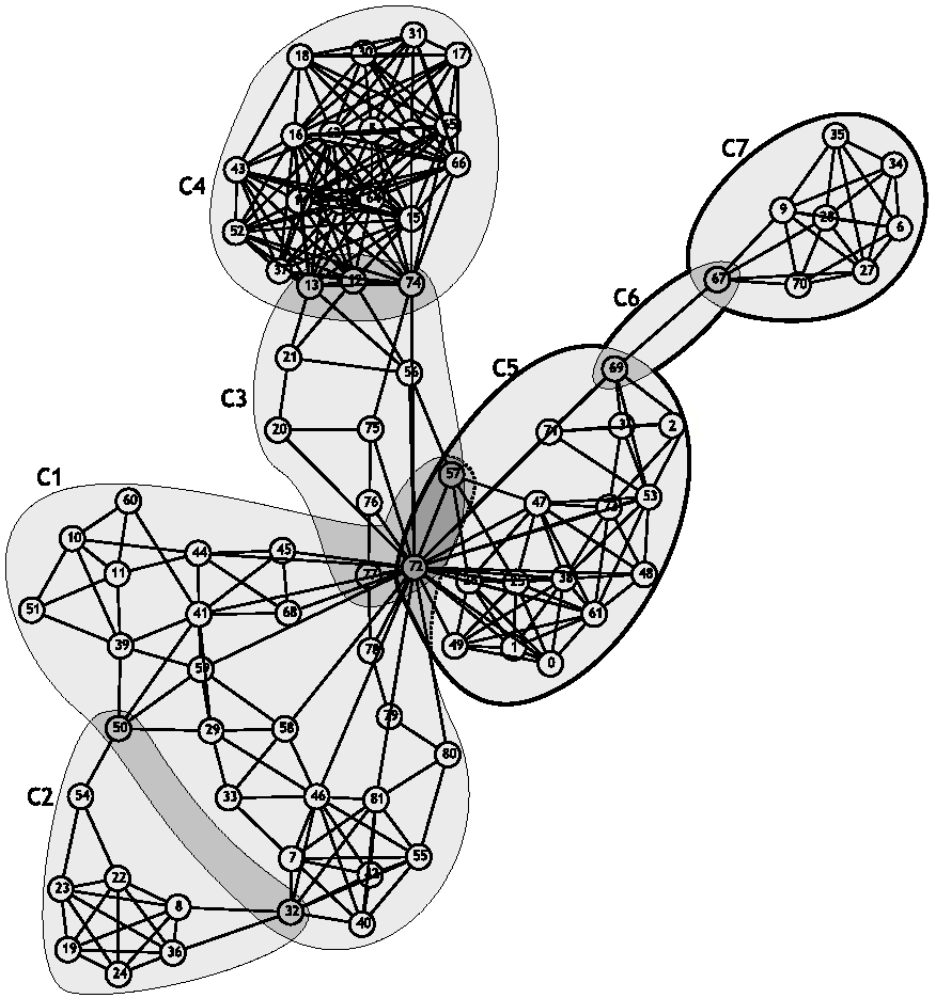


FIGURE 5.1 – *Exemple d'instance structurée provenant du CELAR (figure issue de [DGSV06]).*

— ns est le nombre de sommets de la décomposition arborescente du graphe des contraintes
— p permet de contrôler la densité des contraintes au sein d'un cluster de la décomposition arborescente et est le pourcentage de paires de variables non reliées par une contrainte dans un cluster.

Etant donnés ces paramètres, le générateur d'instances procède de la façon suivante :

— il génère un ensemble X de n variables, et pour toute variable $x_i \in X$, il affecte $D(x_i)$ à $\{1, \ldots, d\}$
— il génère aléatoirement un arbre comportant ns sommets et associe à chaque sommet u de l'arbre un sous-ensemble de variables $X_u \subseteq X$ de telle sorte que $2 \leq |X_u| \leq w$, et pour toute arête (u, v) de l'arbre, $1 \leq |X_u \cap X_v| \leq s$;
— pour chaque sommet u de l'arbre, et pour toute paire de variables $\{x_i, x_j\} \subseteq X_u$, il génère avec la probabilité $1-p$ une contrainte $c_{ij} \in C$ entre x_i et x_j de telle sorte que chaque tuple support de $sup(c_{ij})$ est aléatoirement pris dans $D(x_i) \times D(x_j)$ avec la probabilité $1 - t$.

L'arbre ainsi généré correspond à une décomposition arborescente du CSP (X, D, C) de

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

largeur bornée par w et de taille de séparateurs bornée par s. Notons cependant que cet arbre n'est pas fourni à notre solveur générique, qui utilise l'algorithme décrit dans la section 2.2.4 pour calculer une décomposition arborescente du graphe des contraintes, et il est possible que cette décomposition soit différente de l'arbre initial.

5.2.2 Paramètres retenus pour générer un ensemble d'instances

Les paramètres n et d permettent de contrôler la taille des instances, et nous avons généré des instances avec $n \in \{150, 250, 500\}$ et $d \in \{20, 25\}$. Les paramètres w, s et ns permettent de contrôler la structure des instances, et nous avons généré des instances avec $w \in \{15, 20\}$, $s \in \{5, 10\}$ et $ns \in \{15, 20, 25, 50\}$.

Les paramètres t et p permettent de contrôler la difficulté des instances générées, en faisant varier la densité et la dureté des contraintes. S'il existe des travaux théoriques permettant d'identifier la zone de transition de phase où se trouvent les instances les plus difficiles dans le cas où le générateur d'instances ne force pas les instances à avoir une structure [SD96], ces travaux ne sont pas nécessairement utilisables dans notre contexte où les instances sont structurées. Aussi avons-nous identifié la zone de transition de phase (pour BTD) de façon empirique, en fixant tous les paramètres sauf t (n, d, w, s et ns aux valeurs listées précédemment, et p à une valeur dans l'ensemble $\{0.1, 0.2, 0.3, 0.4\}$), et en faisant varier t. Pour chaque valeur de t, nous avons généré 20 instances différentes, puis nous les avons résolues et compté le nombre d'instances cohérentes et incohérentes, respectivement. La transition de phase correspondant à la zone où la probabilité qu'une instance soit cohérente est égale à $0,5$, nous avons retenu la valeur de p pour laquelle le nombre d'instances cohérentes est proche de la moitié des instances. Nous avons finalement retenu 30 classes d'instances, chaque classe comportant entre 7 et 13 instances. Ces classes vont par paires : chaque paire partage les mêmes paramètres, et permet de distinguer les instances cohérentes des instances incohérentes. Elles sont récapitulées dans le tableau 5.3.

5.3 Discussion

Nous disposons maintenant d'un ensemble d'instances varié de 792 instances provenant d'un benchmark existant, et 300 instances que nous avons générées aléatoirement afin de contrôler leur structure. Ces différentes classes d'instances vont nous permettre de tester notre cadre générique sur des instances aux propriétés diverses.

Nous allons voir au prochain chapitre ce que révèlent les expérimentations.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

classe	n	d	t	w	p	s	ns	#i
c1	150	25	34	15	10	5	15	13
i1	150	25	34	15	10	5	15	7
c2	150	25	38	15	20	5	15	10
i2	150	25	38	15	20	5	15	10
c3	150	25	41	15	30	5	15	13
i3	150	25	41	15	30	5	15	7
c4	150	25	46	15	40	5	15	12
i4	150	25	46	15	40	5	15	8
c5	250	20	25	20	10	5	20	9
i5	250	20	25	20	10	5	20	11
c6	250	20	27	20	20	5	20	10
i6	250	20	27	20	20	5	20	10
c7	250	20	29	20	30	5	20	10
i7	250	20	29	20	30	5	20	10
c8	250	20	32	20	40	5	20	10
i8	250	20	32	20	40	5	20	10
c9	250	20	37	20	10	10	25	9
i9	250	20	37	20	10	10	25	11
c10	250	25	34	15	10	5	25	10
i10	250	25	34	15	10	5	25	10
c11	250	25	37	15	20	5	25	9
i11	250	25	37	15	20	5	25	11
c12	250	25	40	15	30	5	25	8
i12	250	25	40	15	30	5	25	12
c13	250	25	45	15	40	5	25	13
i13	250	25	45	15	40	5	25	7
c14	500	20	30	15	10	5	50	10
i14	500	20	30	15	10	5	50	10
c15	500	20	34	15	20	5	50	10
i15	500	20	34	15	20	5	50	10

Tableau 5.3 – *Les classes d'instances structurées générées. Chaque ligne donne successivement : l'identifiant de la classe – avec c (respectivement i) pour les classes d'instances cohérentes (respectivement incohérentes)–, le nombre de variables (n), la taille des domaines (d), le pourcentage de tuples interdits par contrainte (t), la taille uniforme de la plus grande clique de la décomposition (w), la densité des contraintes au sein d'un cluster (p), la borne maximale sur la tailles des séparateurs (s), le nombre de sommets de la décomposition arborescente (ns) et le nombre d'instances de la classe (#i).*

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Chapitre 6

Résultats expérimentaux

Sommaire

6.1 Évaluation de l'exploitation de décompositions arborescentes avec des approches de retours arrières dynamique 69
6.2 Comparaison avancée de vingt-quatre configurations 72
6.2.1 Protocole expérimental pour la comparaison de configurations 72
6.2.2 Comparaison des taux de succès 74
6.2.3 Comparaison du nombre d'instances « bien résolues » 75
6.2.4 Étude de la répartition en classes des instances bien résolues . . 77
6.2.5 Décomposer ou ne pas décomposer ? 78
6.3 Discussion 79

Notre ensemble d'instances étant désormais déterminé, nous allons pouvoir comparer nos configurations expérimentalement. Nous avons bénéficié du soutien du centre de calcul de l'IN2P3 [1], sans qui cette étude n'aurait pas été réalisable, ni même imaginable. Avec en moyenne 160 exécutions en parallèle pendant plus de trois mois, la puissance de calcul fournie par l'IN2P3 nous a permis ce qui suit.

Nous commencerons par évaluer l'intêret de l'intégration de l'exploitation de la décomposition arbrescente avec des approches de retours arrières dynamique en section 6.1, puis nous continuerons avec une comparaison avancée de vingt-quatre configurations en section 6.2. Le protocole expérimental est décrit succinctement en section 6.2.1. Nous pourrons ensuite analyser les premiers résultats (voir section 6.2.2) et définir ce qu'est une configuration bonne pour une instance donnée (voir section 6.2.3). Cela nous permettra d'identifier les forces des configurations en section 6.2.4 puis d'étudier la pertinence des méthodes de décomposition en section 6.2.5.

6.1 Évaluation de l'exploitation de décompositions arborescentes avec des approches de retours arrières dynamique

Chaque configuration de notre algorithme générique, introduit au chapitre 4, est dénotée par un quadruplet (r, p, h, t) où $r \in \{CBT, CBJ, DBT, CBJR, DR\}$ définit le mécanisme de retour-arrière, $p \in \{FC, MAC\}$ définit le niveau de propagation des contraintes, $h \in \{d, w\}$ définit l'heuristique de choix de variables, et $t \in \{vrai, faux\}$

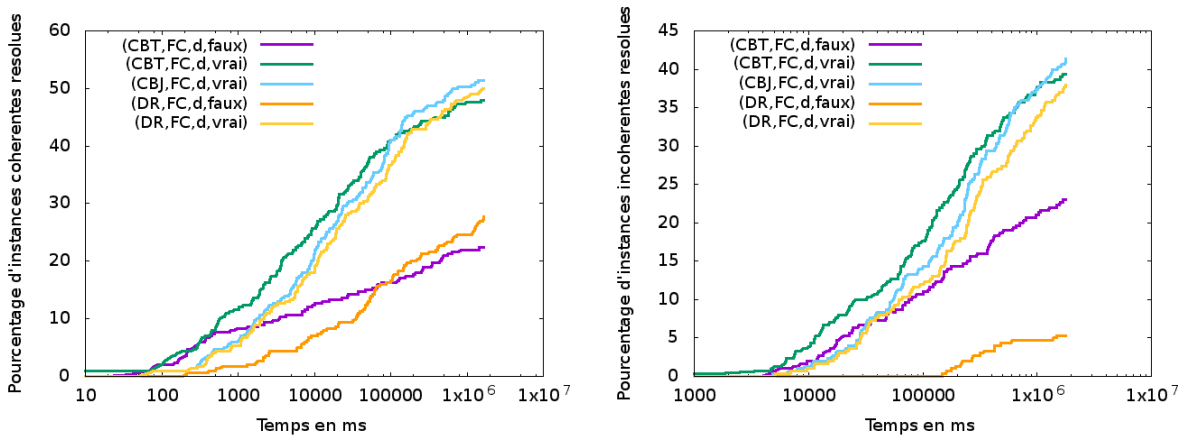
1. http://cc.in2p3.fr/?lang=fr

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

détermine si l'algorithme exploite une décomposition arborescente ou non. Dans cette première section, nous évaluons l'intérêt de combiner une exploitation d'une décomposition arborescente avec trois mécanismes de retour-arrière (CBT, CBJ et DR), les deux autres paramètres étant fixés à $p = FC$ et $h = d$. En effet, l'exploitation d'une décomposition arborescente est souvent combinée avec un retour-arrière chronologique ([JT03b, JNT07b]), et de nombreuses études ont montré tout l'intérêt d'exploiter une décomposition arborescente lorsque les instances ont une petite largeur d'arbre. Les mécanismes de retour-arrière CBJ et DR exploitent de façon dynamique la structure de l'instance (représentée par les explications de conflits) et nous souhaitons étudier l'intérêt de combiner ces mécanismes dynamiques avec une exploitation de la structure statique de l'instance (représentée par la décomposition arborescente).

Nous noterons CBT, CBJ et DR les configurations (CBT,FC,d,faux), (CBJ,FC,d,faux) et (DR,FC,d,vrai), respectivement, et CBT-TD, CBJ-TD et DR-TD les configurations (CBT,FC,d,vrai), (CBJ,FC,d,vrai) et (DR,FC,d,vrai), respectivement.

Dans cette première étude, nous nous concentrons sur les instances qui sont structurées dans le sens où la largeur de la décomposition arborescente calculée par notre algorithme heuristique est inférieure ou égale à la moitié des variables : les 45 instances décrites dans le tableau 5.2, et les 300 instances structurées décrites dans le tableau 5.3. Pour chacune de ces instances, nous avons lancé une exécution de chaque configuration seulement, car il s'agit d'une étude préliminaire visant à décider s'il est intéressant de conserver toutes ces configurations pour l'expérimentation plus intensive qui sera décrite à la section suivante. Nous avons limité chaque exécution à 30 minutes de temps CPU (calcul de la décomposition arborescente compris) sur une machine Intel(R) Xeon(R) CPU E5520 2.27GHz 64 bits, cache 8192 Go, RAM 16 Go.



(a) (b)

FIGURE 6.1 – *Pourcentage d'instances structurées générées aléatoirement (a) cohérentes ou (b) incohérentes résolues au cours du temps par (CBT,FC,d,faux), (CBT,FC,d,vrai), (CBJ,FC,d,vrai), (DR,FC,d,faux) et (DR,FC,d,vrai).*

La figure 6.1 compare CBT et DR avec CBT-TD, CBJ-TD et DR-TD, sur les instances STRUCT cohérentes (à gauche) et incohérentes (à droite) décrites dans le tableau 5.3. Pour les instances cohérentes, nous constatons que les trois méthodes exploitant une décomposition arborescente (CBT-TD, DR-TD et CBJ-TD) ont des résultats très proches, et que ces résultats sont bien meilleurs que ceux obtenus par les méthodes n'exploi-

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

	CBT		CBT-TD		$\frac{\text{CBT}}{\text{CBT-TD}}$	CBJ		CBJ-TD		$\frac{\text{CBJ}}{\text{CBJ-TD}}$	DR		DR-TD		$\frac{\text{DR}}{\text{DR-TD}}$
	T	(KN)	T	(KN)	T	T	(KN)	T	(KN)	T	T	(KN)	T	(KN)	T
an8	4.6	(231)	4.8	(291)	~ 1	10.9	(231)	10.7	(231)	~ 1	11.6	(251)	12.7	(291)	~ 1
an9	54.4	(4434)	1346.8	(77286)	.04	189.6	(4433)	317.7	(10747)	.6	223.9	(4948)	615.6	(12685)	.4
da8	0.9	(110)	0.8	(110)	~ 1	2.7	(110)	2.6	(110)	~ 1	2.7	(110)	2.8	(110)	~ 1
da9	6.6	(986)	6.1	(986)	~ 1	24.9	(986)	24.7	(986)	~ 1	24.4	(986)	24.7	(986)	~ 1
da10	54.5	(9864)	49.4	(9864)	~ 1	256.7	(9864)	251.5	(9864)	~ 1	253.2	(9864)	259.3	(9864)	~ 1
ho8	13.7	(109)	13.4	(109)	~ 1	25.7	(109)	25.2	(109)	~ 1	24.4	(109)	25.7	(109)	~ 1
hu8	235.6	(79204)	0.3	(109)	**873**	1.9	(118)	1.7	(109)	**1.14**	2.1	(118)	1.8	(109)	**1.2**
hu9	-		1.95	(986)	>**1000**	17.0	(1001)	16.6	(986)	~ 1	18.6	(1002)	17.9	(986)	~ 1
hu10	-		17.3	(9864)	>**208**	-		174.4	(9864)	>**20**	-		183.8	(9864)	>**19**
j7	2.7	(539)	0.1	(13)	**44**	0.8	(43)	0.3	(14)	**3.3**	0.8	(43)	0.2	(14)	**3.5**
j8	37.7	(9219)	0.4	(110)	**97**	8.5	(443)	2.1	(110)	**4.2**	9.0	(443)	2.1	(110)	**4.4**
j9	575.5	(152918)	3.2	(986)	**179**	100.9	(4939)	19.8	(986)	**5.1**	105.5	(4939)	19.9	(986)	**5.3**
m1	-		0.0	(1)	>**1000**	0.6	(86)	0.0	(1)	**62**	-		0.0	(1)	>**1000**
m2	-		0.0	(1)	>**1000**	0.2	(23)	0.0	(1)	**15**	-		0.0	(1)	>**1000**
m3	-		0.0	(1)	>**1000**	0.3	(32)	0.0	(1)	**26**	-		0.0	(1)	>**1000**
m4	-		0.0	(1)	>**1000**	0.2	(19)	0.0	(1)	>**1000**	-		0.0	(1)	>**1000**
ha7	6.1	(1)	6.6	(1)	~ 1	6.2	(1)	7.0	(1)	.89	6.9	(1)	6.5	(1)	~ 1
gm2	9.1	(297)	-		<.001	0.2	(1)	-		<.001	-		-		-
gm8	-		-		-	-		18.9	(49)	>**190**	-		-		-
gm9	-		-		-	29.7	(51)	-		<.01	-		-		-
gm12	10.6	(79)	-		.003	0.1	(0)	-		<.001	-		-		-
gm13	-		-		-	0.2	(0)	-		<.001	0.97	(1)	-		<.001
g1	0.0	(0)	0.0	(1)	.5	0.0	(0)	0.0	(0)	.67	0.0	(0)	1.4	(7)	.007
g2	0.0	(1)	-		<.001	0.1	(1)	-		<.001	0.1	(1)	-		<.001
g3	0.0	(0)	-		<.001	0.0	(0)	-		<.001	56.7	(293)	-		<.016
g8	0.1	(1)	-		<.001	0.2	(1)	-		<.001	0.1	(1)	-		<.001
g9	0.1	(1)	-		<.001	0.3	(1)	-		<.001	0.3	(1)	-		<.001
g14	0.1	(1)	112.3	(1241)	.001	0.2	(1)	1134.1	(1169)	<.001	0.2	(1)	1373.2	(1417)	<.001
sm1	-		1.2	(12)	>**1000**	0.1	(1)	2.3	(7)	0.034	7.7	(29)	0.4	(2)	**21**
sm2	-		-		-	10.9	(107)	-		<.003	-		-		-
sm3	-		2.5	(50)	>**1000**	7.1	(52)	6.7	(48)	~ 1	-		-		-
sm4	0.0	(5)	0.9	(120)	.05	0.1	(4)	0.2	(8)	.5	-		-		-
sm5	-		6.7	(706)	>**537**	0.1	(1)	10.1	(166)	.007	-		-		-
sm6	-		0.0	(0)	>**1000**	0.1	(0)	0.2	(0)	.7	-		0.1	(0)	>**1000**
sm7	0.7	(40)	0.2	(12)	**3**	0.4	(4)	0.2	(4)	**1.7**	-		-		-
sm8	-		4.0	(111)	>**905**	3.5	(35)	3.3	(32)	~ 1	-		-		-
sm9	0.0	(1)	0.0	(1)	.5	0.0	(1)	0.0	(1)	~ 1	-		0.0	(1)	>**1000**
sm10	0.0	(1)	0.0	(1)	~ 1	0.0	(1)	0.0	(1)	**2**	-		0.0	(1)	>**1000**
s1	0.0	(1)	9.8	(50)	.003	0.0	(1)	2.3	(4)	0.02	0.1	(1)	-		<.001
s3	0.0	(0)	-		<.001	0.0	(0)	-		<.001	0.0	(0)	-		<.001
s4	-		3.4	(21)	>**1000**	-		3.4	(5)	>**1000**	-		7.8	(12)	>**459**
s5	0.1	(1)	0.5	(2)	.17	0.1	(1)	0.5	(1)	.18	229.8	(495)	0.7	(1)	**338**
s11	26.3	(169)	-		<.007	10.9	(14)	-		<.003	1398.5	(1736)	-		<.38
su6	0.0	(1)	0.0	(1)	**1.5**	0.0	(1)	0.0	(1)	~ 1	-		-		-
su7	0.0	(0)	0.0	(0)	.75	0.0	(0)	0.0	(0)	~ 1	0.0	(1)	-		<.001

Tableau 6.1 – *Résultats sur les instances ACAD, PATT et REAL structurées. Les six configurations sont CBT, BTD, CBJ, CBJ-TD, DR, et DR-TD avec FC et ddeg. T donne le nombre de secondes pour résoudre l'instance (ou - si l'instance n'est pas résolue en une heure) et (KN) donne le nombre de milliers de nœuds.* $\frac{CB}{CBT\text{-}TD}$ *(respectivement* $\frac{CBJ}{CBJ\text{-}TD}$ *et* $\frac{DR}{DR\text{-}TD}$*) donnent l'amélioration en temps apportée par l'exploitation de la décomposition pour CBT (respectivement CBJ et DR).*

tant pas de décomposition arborescente. Au temps limite de 30 minutes, la configuration ayant résolu le plus d'instances est CBJ-TD. Pour les instances incohérentes, les méthodes exploitant une décomposition arborescente ont des résultats relativement proches aussi, mais DR-TD est moins bonne que CBT-TD et CBJ-TD. De fait, DR est généralement moins efficace que CBT ou CBJ pour prouver l'incohérence d'instances.

Le tableau 6.1 compare nos six configurations (CBT, CBJ, DR, CBT-TD, CBJ-TD et DR-TD) sur chaque instance structurée retenue du benchmark de la compétition 2008. Pour chaque instance, et pour chaque retour-arrière $r \in \{CBT, CBJ, DR\}$, il donne le ratio de temps entre r et r-TD : lorsque ce ratio est supérieur à 1, la décomposition

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

arborescente améliore la résolution ; quand il est proche de 1 elle ne change rien ; quand il est inférieur à 1 elle dégrade. Pour les instances de la classe PATT, la décomposition arborescente améliore les résultats, ou ne les change pas, sauf pour une instance (an9), qui est toujours mieux résolue sans exploiter de décomposition arborescente. Pour les instances des classes ACAD et REAL, les résultats sont plus mitigés : si le ratio est parfois supérieur à 1 (notamment pour CBT), il est aussi bien souvent très inférieur à 1 et pour de nombreuses instances, l'exploitation d'une décomposition arborescente dégrade la résolution.

Pour quasiment toutes les instances du tableau 6.1, CBT-TD est meilleure que CBJ-TD et DR-TD : combiner des retour-arrières intelligents avec une exploitation de décomposition arborescente ne semble pas améliorer la résolution. Cela peut probablement s'expliquer par le fait que les mécanismes d'exploitation dynamique de la structure utilisés par CBJ et DR sont redondants avec ceux utilisant la structure statique. Ainsi, cette première série d'expérimentations nous a amenés à abandonner les configurations (CBJ,*,*,vrai) et (DR,*,*,vrai) dans les expérimentations intensives décrites dans la section suivante.

6.2 Comparaison avancée de vingt-quatre configurations

Nous allons maintenant comparer les vingt-quatre configurations qu'il nous reste sur notre ensemble d'instances.

Dans la mesure où seules les configurations utilisant CBT comme mécanisme de retour-arrière peuvent être combinées avec une exploitation d'une décomposition arborescente, nous supprimons le quatrième paramètre pour alléger les notations. Ainsi, pour $r \in \{CBJ, DBT, DR\}$, $p \in \{FC, MAC\}$, et $h \in \{d, w\}$, nous noterons (r, p, h) la configuration $(r, p, h, faux)$ (et la configuration $(r, p, h, vrai)$ n'est pas évaluée). Pour $p \in \{FC, MAC\}$, et $h \in \{d, w\}$, nous noterons (CBT,p,h) la configuration (CBT,p,h,faux), et (BTD,p,h) la configuration (CBT,p,h,vrai) (BTD étant le nom initialement introduit par les auteurs de [JT03b]).

6.2.1 Protocole expérimental pour la comparaison de configurations

Les exécutions ont été réalisées sur les machines de l'IN2P3 sur des Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz, 20480 KB de cache, 3GB RAM. Comme toutes les configurations sont non déterministes (voir la section 4.5.2), nous avons lancé chaque configuration quinze fois sur chaque instance. Chaque exécution a été limitée à trente minutes de temps CPU et 3GB de mémoire. Si l'exécution réussit à trouver une solution ou à prouver l'incohérence dans ces limites, alors nous disons qu'elle a réussi, sinon nous disons qu'elle a échoué.

Pour comparer les résultats de deux configurations sur une même instance, nous procédons de la sorte. Pour chaque instance i de notre benchmark et pour chaque configuration c de notre cadre, nous notons $E_{c,i}$ le nombre d'échecs sur les quinze essais de c sur i, et $T_{c,i}$ la collection de $15 - E_{c,i}$ temps d'exécution des essais ayant réussi, chaque essai ayant été réalisé avec une graine différente pour le générateur de nombres aléatoires.

Pour chaque instance i, nous choisissons de définir la meilleure configuration comme étant celle qui minimise en priorité le nombre d'exécutions échouées, puis –pour départager les égalités– celle qui minimise la moyenne sur i des temps d'exécutions : $best_i = argmin_c(\text{ordreLexicographique}(E_{c,i}, \text{moyenne}(T_{c,i})))$.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

essai	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
(CBT,FC,d)	990	960	990	930	1010	980	700	1040	1060	1010	990	980	970	990	920
(CBT,FC,w)	890	960	890	920	990	1150	1010	960	770	1040	750	690	1010	1000	880
(CBT,MAC,d)	1240	1770	1690	1760	1690	1620	1750	1510	1810	1270	1810	1830	1750	1530	1400
(CBT,MAC,w)	1680	1690	1710	1560	1660	1820	1760	1290	1230	1800	1770	1720	1330	1420	1680
(DBT,FC,d)	2850	3100	3360	2740	3270	3110	2730	2350	3470	3520	2570	3570	3210	3630	3460
(DBT,FC,w)	3560	7140	2970	3210	3280	2700	2570	3350	3110	3370	2410	3260	2770	3140	2570
(DBT,MAC,d)	5350	5720	6240	4380	4870	5600	5510	4140	5610	5780	6080	6020	5350	5590	5650
(DBT,MAC,w)	5570	5520	5500	5630	5660	5680	4270	5600	5450	5960	5510	5550	5940	4570	5440
(CBJ,FC,d)	3460	3260	3150	2740	2780	3090	3120	2410	3180	3420	3370	3040	3050	3250	3080
(CBJ,FC,w)	3330	3730	3980	2810	3260	3050	3280	3200	2330	3430	2350	3180	2920	3210	3490
(CBJ,MAC,d)	4810	5560	5460	5710	5600	5580	6370	3770	5490	4390	4790	5290	6090	4340	4300
(CBJ,MAC,w)	5460	5610	5450	5470	5620	5660	5530	6090	3670	5090	4420	5370	3860	5300	5480
(CBJR,FC,d)	3960	3720	3540	3970	3970	3840	3290	3020	4190	3840	4230	4080	4270	3880	2890
(CBJR,FC,w)	3250	4120	4270	3920	3780	3970	4400	3950	3360	3970	4170	3650	3920	4180	4100
(DR,FC,d)	3280	4510	5310	5840	3670	4520	4960	3520	3380	4010	3440	3630	4130	3860	1850
(DR,FC,w)	5050	5650	4350	4910	4130	4240	5390	5080	5350	5310	5430	4970	4470	4850	5200
(DR,MAC,d)	13200	11610	14510	11110	15970	11070	15670	13650	14350	11910	13090	13350	10970	14830	14390
(DR,MAC,w)	14320	14770	14980	14270	11590	16190	15080	12010	14030	12230	12580	16780	12840	12360	12880
(BTD,FC,d)	1010	1030	1040	940	830	950	1040	840	820	1060	1080	780	1120	1010	930
(BTD,FC,w)	1070	980	930	850	970	1100	730	1040	790	930	700	1170	970	1020	860
(BTD,MAC,d)	1780	1870	1670	1790	1750	1810	1620	1710	1760	1850	1850	1810	1730	1640	1750
(BTD,MAC,w)	1860	1760	1850	1410	1430	1200	1810	1840	1730	1860	1570	1600	1790	1830	1410

Tableau 6.2 – *Résultats de vingt-deux configurations (temps en milli-secondes) sur l'instance pigeon10 dans ACAD. Nous colorons le meilleur temps pour chaque exécution.*

Considérons par exemple l'instance pigeon10 de notre benchmark. Le tableau 6.2 donne les temps de chacune des 15 exécutions. Notons que si nous n'avions considéré qu'une seule exécution, alors la meilleure exécution aurait été différente selon le run considéré : par exemple, à l'exécution 4 la meilleure configuration est (BTD,FC,w), à l'exécution 3, c'est (CBT,FC,w), à l'exécution 5, c'est (BTD,FC,d) et à l'exécution 7 c'est (CBT,FC,d). Le tableau 6.3 donne pour chaque configuration, le temps minimal, moyen, médian et maximal, parmi les 15 exécutions. Sur ce tableau, nous voyons que $best_pigeon10 = (CBT, FC, w)$.

Cependant, nous voyons bien que certaines configurations ont des résultats qui semblent assez proches de ceux de (CBT,FC,w). Nous utilisons donc un test statistique pour vérifier, pour chacune des configurations $c \neq$(CBT,FC,w), l'hypothèse suivante : est-ce que (CBT,FC,w) est significativement meilleure que c pour cette instance ? Le test que nous utilisons est le test de Student, qui évalue la probabilité que les deux collections $T_{c,pigeon10}$ et $T_{(CBT,FC,w),pigeon10}$ proviennent de deux variables aléatoires ayant une même espérance. Nous avons fixé le seuil de risque p à $0,01$.

Sur notre exemple, le test de Student avec $p = 0,01$ nous permet de décider que (CBT,FC,w) n'est pas significativement meilleur que les configurations (CBT,FC,d), (BTD,FC,d) et (BTD,FC,w) alors qu'il est significativement meilleur que toutes les autres configurations.

Plus généralement, nous dirons qu'une configuration c' est bonne pour une instance i soit si $c' = best_i$, soit si $E_{c',i} = E_{best_i,i}$ et c' n'est pas significativement moins bonne que $best_i$ selon un test de Student avec un seuil de risque p à $0,01$.

Pour pigeon10, les bonnes configurations sont donc (CBT,FC,w), (CBT,FC,d), (BTD,FC,w) et (BTD,FC,d).

Notons que 15 valeurs pour un test statistique est probablement une limite basse et nous aurions obtenu des résultats plus robustes avec un plus grand nombre d'exécutions. Cependant, nous avons été limités par nos ressources (bien que nous ayons été aidés par le centre de calcul de l'IN2P3).

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

	min	moy	méd	max
(CBT,FC,d)	700	968,0	990	1060
(CBT,FC,w)	690	927,3	960	1150
(CBT,MAC,d)	1240	1628,7	1690	1830
(CBT,MAC,w)	1230	1608,0	1680	1820
(CBJ,FC,d)	2350	3129,3	3210	3630
(CBJ,FC,w)	2410	3294,0	3140	7140
(CBJ,MAC,d)	4140	5459,3	5600	6240
(CBJ,MAC,w)	4270	5456,7	5550	5960
(CBJR,FC,d)	2410	3093,3	3120	3460
(CBJR,FC,w)	2330	3170,0	3210	3980
(DBT,FC,d)	3770	5170,0	5460	6370
(DBT,FC,w)	3670	5205,3	5460	6090
(DBT,MAC,d)	2890	3779,3	3880	4270
(DBT,MAC,w)	3250	3934,0	3970	4400
(DR,FC,d)	1850	3994,0	3860	5840
(DR,FC,w)	4130	4958,7	5050	5650
(DR,MAC,d)	10970	13312,0	13350	15970
(DR,MAC,w)	11590	13794,0	14030	16780
(BTD,FC,d)	780	965,3	1010	1120
(BTD,FC,w)	700	940,7	970	1170
(BTD,MAC,d)	1620	1759,3	1760	1870
(BTD,MAC,w)	1200	1663,3	1760	1860

Tableau 6.3 – *Résultats des vingt-deux configurations (temps en milli-secondes) sur l'instance pigeon10 dans ACAD. Nous donnons le temps minimum, la moyenne, la médiane et le maximum sur quinze essais.*

6.2.2 Comparaison des taux de succès

Le tableau 6.4 compare le pourcentage d'exécutions réussies pour chacune de nos vingt-quatre configurations sur les 1092 instances de notre benchmark, pour la limite de trente minutes (1800 secondes), mais aussi pour d'autres limites de temps intermédiaires, permettant de mesurer l'évolution de ce pourcentage dans le temps. Nous donnons également les taux de succès de Gecode (avec le modèle proposé dans [MMG11]) avec trois niveaux de propagations de contraintes différents : ICL_VAL, ICL_DOM et ICL_DEF. Cela nous montre que notre implémentation est compétitive avec Gecode. Bien sûr, notre implémentation est dédiée uniquement aux CSP binaires, là où Gecode est un solveur qui peut également résoudre des CSP n-aires, utiliser les contraintes globales… et n'est donc pas spécialisé pour les CSPs binaires. Cette comparaison montre simplement que notre base de code est compétitive avec l'état de l'art et justifie donc son utilisation pour cette étude.

La meilleure configuration pour toutes limites de temps considérées est (CBT,MAC,w) si nous considérons le taux de succès global. Ce résultat n'est pas surprenant et a déjà été observé, par exemple dans [LBH04].

Sans surprise, nous notons aussi que les configurations qui utilisent wdeg comme heuristique de choix de variable ont de meilleures performances que les configurations utilisant ddeg, comme vu dans [BHLS04]. Cependant le gain dépend du mécanisme de retour arrière considéré, comme vu dans [CvB01]. En particulier, utiliser wdeg améliore grandement le processus de résolution pour CBT, DBT et DR tandis que l'amélioration

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

			1	5	10	50	100	500	1000	1800
CBT	FC	d	37.0	45.2	47.6	52.7	55.3	60.0	61.1	61.7
		w	41.8	51.7	56.8	65.9	69.4	77.8	81.5	83.2
	MAC	d	43.0	51.7	56.7	65.5	69.1	75.3	76.5	77.7
		w	**47.1**	**61.5**	**68.3**	**80.5**	**85.2**	**92.3**	**94.3**	**95.4**
CBJ	FC	d	**41.3**	50.4	55.2	66.9	70.5	81.9	85.6	88.0
		w	39.6	51.0	55.0	67.8	72.6	84.0	88.1	91.0
	MAC	d	38.0	50.2	54.3	68.2	74.2	85.3	88.8	90.4
		w	39.7	**53.1**	**57.6**	**73.7**	**79.6**	**90.7**	**93.5**	**95.1**
CBJR	FC	d	**39.9**	49.4	53.3	63.3	66.9	75.8	78.1	79.5
		w	39.1	**50.5**	**55.0**	**67.5**	**72.6**	**84.2**	**88.3**	**90.9**
	MAC	d	29.2	37.3	41.1	46.4	48.5	53.9	55.4	56.5
		w	31.6	40.1	44.9	55.0	58.9	67.7	69.4	70.7
DBT	FC	d	33.8	38.0	38.8	40.8	41.5	43.9	45.8	46.7
		w	37.7	47.4	50.5	61.9	66.5	77.0	80.3	83.7
	MAC	d	35.8	46.2	49.4	56.6	60.0	66.6	68.0	69.3
		w	**37.9**	**49.5**	**54.1**	**68.6**	**74.5**	**85.7**	**89.5**	**91.8**
DR	FC	d	32.7	37.5	39.2	41.9	42.6	44.0	44.6	45.0
		w	**35.1**	**44.4**	48.1	55.4	59.8	71.9	76.3	79.4
	MAC	d	32.5	41.6	45.1	51.9	53.8	59.0	60.8	62.3
		w	34.4	44.3	**48.7**	**57.8**	**62.5**	**75.2**	**80.3**	**84.5**
BTD	FC	d	31.4	45.1	52.2	65.1	69.2	76.1	77.3	78.0
		w	33.5	48.0	55.5	70.2	74.9	82.1	83.9	84.5
	MAC	d	32.8	45.1	53.9	70.1	75.8	84.6	86.0	87.1
		w	**37.4**	**51.3**	**61.9**	**77.6**	**83.3**	**91.9**	**93.6**	**94.2**
Gecode	ICL_DEF		29.7	34.9	38.1	48.9	55.4	**66.7**	69.3	71.9
Gecode	ICL_VAL		27.7	32.9	35.2	45.3	51.3	63.8	67.2	70.4
Gecode	ICL_DOM		**29.9**	**35.8**	**38.9**	**50.9**	**56.6**	**66.7**	**70.5**	**73.4**

Tableau 6.4 – *Comparaison selon le taux de succès global. Chaque ligne affiche successivement la configuration du cadre générique et le pourcentage d'essais réussis à différentes limites de temps CPU (en secondes, sur 15 essais sur 1092 instances). Pour chaque stratégie de retour arrière nous mettons en gras la meilleure configuration. Nous colorons en bleu la meilleure configuration de toutes.*

n'est pas énorme pour CBJ.

Par ailleurs, les configurations utilisant DBT ou DR comme mécanisme de retour arrière et FC comme niveau de propagation de contraintes ont de mauvaises performances. Nous avions déjà vu dans [Bak94] que DBT pouvait obtenir de mauvaises performances.

6.2.3 Comparaison du nombre d'instances « bien résolues »

Ces taux de succès globaux sur les 1092 instances de notre ensemble d'instances cachent des résultats très différents lorsque nous regardons chaque instance séparément. En particulier, certaines configurations qui ont un taux de succès global plutôt faible sur l'ensemble des instances sont les meilleures configurations sur certaines instances.

La ligne (m) du tableau 6.5 affiche le pourcentage d'instances pour lesquelles une

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

	CBT				CBJ				CBJR			
	FC		MAC		FC		MAC		FC		MAC	
	d	w	d	w	d	w	d	w	d	w	d	w
(m)	**27.7**	4.5	11.3	9.2	2.0	1.2	1.0	0.7	0.7	1.1	1.6	0.9
(m/d)	17.8	2.3	8.8	11.1	1.1	0.3	0.3	1.1	0.8	1.6	0.0	0.8
(b/d)	27.7	9.8	12.9	19.9	4.0	3.4	2.7	7.1	2.3	4.8	2.3	2.3
(sm/d)	6.4	0.3	5.5	5.8	0	0	0.2	0.5	0	1.1	0	0
(sm2/d)	6.4	0.6	6.1	7.1	1.6	-	0.2	0.8	-	1.4	-	-

	DBT				DR				BTD			
	FC		MAC		FC		MAC		FC		MAC	
	d	w	d	w	d	w	d	w	d	w	d	w
(m)	1.4	0.4	0.7	0.1	1.0	3.4	0.5	0.2	14.2	12.3	0.7	3.1
(m/d)	0.8	0.2	0.5	0.0	0.5	2.7	0.5	0.2	**23.3**	18.8	1.3	5.1
(b/d)	1.3	5.5	2.7	2.7	2.7	7.1	2.9	2.9	**42.1**	26.5	5.3	10.5
(sm/d)	0	0	0	0	0	1.9	0	0	**14.0**	7.2	0.2	1.3
(sm2/d)	-	-	-	-	-	2.3	-	-	**14.6**	7.9	0.3	1.3

Tableau 6.5 – *Pour chaque configuration c, la ligne (m) donne le pourcentage d'instances pour lequel c est la meilleure configuration ; la ligne (m/d) (respectivement (b/d)) donne le pourcentage d'instances difficiles pour lequel c est la meilleure configuration (respectivement c est une bonne configuration) ; la ligne (sm/d) (respectivement (sm2/d)) donne le pourcentage d'instances difficiles pour lequel c est la meilleure configuration et toutes les autres configurations sont significativement plus mauvaises que c (respectivement que toutes les autres configurations sauf (CBJ,FC,w), (CBJR,FC,d), (CBJR,MAC,*), (DBT,*,*), (DR,FC,d) et (DR,MAC,*) sont significativement plus mauvaises que c).*

configuration est la meilleure parmi l'ensemble des vingt-quatre configurations. Cela nous montre que même si (CBT,FC,d) ne résout que 61,7% des instances après 30 minutes de temps CPU, c'est la meilleure configuration pour 27,7% des 1092 instances. Bien sur, il est bien connu que des configurations simples comme (CBT,FC,d) ont de meilleures performances que des configurations plus complexes sur des instances très faciles – où il n'y a pas besoin de mécanismes intelligents mais coûteux – tandis que ces configurations simples auront de très mauvaises performances sur des instances plus dures.

Sur la ligne (m/d) du tableau 6.5, nous avons enlevé les instances faciles de l'ensemble d'instances : nous considérons qu'une instance est *facile* si elle est résolue en moins d'une seconde par (CBT,MAC,w) sur chacun des quinze essais. Avec cette définition, 470 instances de notre ensemble d'instances sont faciles et 622 sont plus difficiles. En se concentrant sur ces instances plus difficiles, la ligne (m/d) du tableau 6.5 nous montre que certaines configurations, comme celles utilisant DBT ou DR comme mécanisme de retour arrière, n'ont que très peu d'instances où ce sont les meilleures configurations et encore moins d'instances difficiles où c'est encore le cas.

La meilleure configuration à la ligne précédente (m) était (CBT,FC,d) avec 27,7% d'instances ; à la ligne (m/d) cette configuration n'a plus que 17,8% d'instances où elle est la meilleure. Cette configuration est donc particulièrement efficace sur des instances faciles. Notons également que pour les mécanismes de retour arrière CBT, CBJ et BTD, la variante MAC avec wdeg est proportionnellement meilleure sur les instances difficiles que sur les instances faciles, là où les trois autres variantes sont moins bonnes. Le coût de mise en place de MAC et wdeg est ainsi plus facilement rentabilisé sur des instances plus dures. Quant à BTD, c'est le seul mécanisme de retour arrière dont les

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

quatre variantes sont proportionnellement meilleures sur les instances difficiles que sur les instances faciles ; notamment (BTD,FC,d) est la meilleure configuration pour le plus grand nombre d'instances difficiles.

Comme de nombreuse configurations peuvent avoir des résultats proches pour certaines instances, nous étudions aussi le nombre d'instances pour lequel une configuration a de « bons » résultats en utilisant le test de Student comme expliqué précédemment en section 6.2.1.

La ligne (b/d) du tableau 6.5 affiche le pourcentage d'instances difficiles pour lesquelles une configuration est bonne. Encore une fois, nous notons que les configurations bonnes pour beaucoup d'instances n'ont pas toujours un taux de succès global élevé sur l'ensemble des instances. En particulier, les deux configurations qui sont bonnes pour le plus grand nombre d'instances (à savoir (BTD,FC,d) et (CBT,FC,d)) sont loin d'avoir le meilleur taux de succès global dans le tableau 6.4. Notons que (BTD,FC,d) est une configuration bonne pour plus de 40% des instances, ce qui indique la versatilité de cette configuration. Par ailleurs (CBT,FC,d), la deuxième meilleure configuration sur ce critère, est bonne sur presque 15% d'instances de moins que (BTD,FC,d).

Toutes les configurations sont bonnes pour au moins une instance de l'ensemble d'instances. Cependant, il se peut qu'une configuration soit bonne seulement sur des instances pour lesquelles d'autres configurations sont bonnes, c'est-à-dire que certaines configurations sont dominées par d'autres. Pour étudier cet aspect, la ligne (sm/d) du tableau 6.5 affiche le pourcentage d'instances difficiles pour lesquelles une configuration est la meilleure et toutes les autres configurations sont significativement plus mauvaises. Cela nous montre que douze configurations sont dominées par les autres configurations : (CBJ,FC,*), (CBJR,FC,d), (CBJR,MAC,*), (DBT,*,*), (DR,FC,d) et (DR,MAC,*). Donc, nous avons retiré ces configurations de notre étude, sauf (CBJ,FC,d) : il y a dix instances pour lesquelles toutes les bonnes configurations appartiennent au douze configurations dominées ; comme (CBJ,FC,d) est bonne sur ces dix instances et que c'est la seule configuration parmi les douze dominées à être bonne sur ces dix instances, nous gardons (CBJ,FC,d).

Enfin, la ligne (sm2/d) du tableau 6.5 donne le pourcentage d'instances difficiles pour lesquelles une configuration est la meilleure et toutes les autres configurations sauf (CBJ,FC,*), (CBJR,FC,d), (CBJR,MAC,*), (DBT,*,*), (DR,FC,d) et (DR,MAC,*) sont significativement plus mauvaises. L'ensemble composé des treize configurations restantes contient une configuration bonne pour chaque instance. Cet ensemble est minimal car chacune de ces treize configurations est la seule meilleure pour au moins une instance.

6.2.4 Étude de la répartition en classes des instances bien résolues

Il est intéressant de regarder quelles sont les instances où chacune des treize configurations restante est significativement meilleure que les douze autres configurations. Ces données sont présentes dans le tableau 6.6.

Nous nous apercevons par exemple que (BTD,FC,d) n'est seule meilleure que sur les instances de type STRUCT. Bien que le *Forward Checking* et l'heuristique de choix de variable ddeg ne soient pas considérés comme les plus efficaces, la combinaison avec BTD comme mécanisme de retour arrière offre une méthode significativement meilleure, pour les instances STRUCT.

Nous avons remarqué quelques autres spécificités. La configuration (BTD,FC,w) est significativement meilleure sur 24 instances de type STRUCT et 14 instances de type RAND entre autres, mais de façon plus surprenante, c'est aussi la configuration qui

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

	ACAD	PATT	QRND	RAND	REAL	STRUCT	total
(CBT,FC,d)	5 (0)	30 (17)	4 (0)	117 (17)	12 (5)	9 (7)	177 (46)
(CBT,FC,w)	4 (0)	9 (1)	2 (0)	41 (4)	10 (0)	0 (0)	66 (5)
(CBT,MAC,d)	7 (0)	27 (4)	0 (0)	15 (0)	48 (31)	17 (10)	114 (45)
(CBT,MAC,w)	9 (0)	27 (4)	12 (7)	16 (0)	31 (10)	44 (29)	139 (50)
(CBJ,FC,d)	3 (0)	10 (7)	0 (0)	18 (0)	8 (1)	5 (4)	44 (12)
(CBJ,MAC,d)	0 (0)	2 (0)	0 (0)	14 (0)	16 (0)	2 (1)	34 (1)
(CBJ,MAC,w)	0 (0)	10 (0)	5 (0)	14 (0)	24 (4)	5 (1)	58 (5)
(CBJR,FC,w)	3 (0)	15 (3)	1 (0)	15 (0)	7 (0)	5 (4)	46 (7)
(DR,FC,w)	8 (4)	15 (7)	0 (0)	10 (0)	11 (2)	0 (0)	44 (13)
(BTD,FC,d)	3 (0)	9 (0)	4 (0)	97 (0)	1 (0)	164 (100)	278 (100)
(BTD,FC,w)	12 (7)	10 (5)	4 (2)	60 (14)	7 (1)	85 (24)	178 (53)
(BTD,MAC,d)	5 (0)	8 (0)	0 (0)	15 (0)	2 (0)	21 (2)	51 (2)
(BTD,MAC,w)	9 (0)	26 (3)	0 (0)	16 (0)	8 (3)	28 (6)	87 (12)

Tableau 6.6 – *Nombre d'instances pour lesquelles une configuration est bonne pour chaque classe d'instances (et entre parenthèse nombre d'instances pour lesquelles la configuration est la seule meilleure). Les classes ACAD, PATT, QRND, RAND et REAL sont décrites en section 5.1 ; la classe STRUCT regroupe toutes les instances décrites en section 5.2.*

est significativement meilleure que les autres sur 7 instances provenant de la catégorie ACAD. La seule autre configuration à être significativement meilleure que les autres pour des instances provenant de la catégorie ACAD est (DR,FC,w) et uniquement pour des instances du problème *queensKnights*.

Enfin, la configuration (CBT,MAC,d) est sur beaucoup d'instances de type *fapp* de la classe REAL significativement meilleure que les douze autres configurations.

6.2.5 Décomposer ou ne pas décomposer ?

Le mécanisme BTD est très efficace sur beaucoup d'instances. En particulier, (BTD,FC,d) est bonne sur plus de 42% des instances difficiles et elle est significativement meilleure que toutes les autres configurations sur plus de 14% des instances difficiles. Cependant, sur d'autres instances elle a de très mauvaises performances de telle sorte que son taux de succès global est plutôt bas comparé à d'autres approches. Afin de comprendre quelles instances sont mieux résolues par BTD, nous partitionnons les 622 instances difficiles en trois ensembles :

— l'ensemble *Décompose* contient toutes les instances difficiles qui sont le mieux résolues par une des quatre configurations basées sur BTD (à savoir (BTD,*,*)) et pour lesquelles aucune des neuf configurations non basées sur BTD n'est bonne (il reste (CBT,*,*), (CBJ,FC,d), (CBJ,MAC,*), (CBJR,FC,w) et (DR,FC,w)) ;

— l'ensemble *Décompose pas* contient toutes les instances difficiles qui sont le mieux résolues par les neuf configurations non basées sur BTD et pour lesquelles aucune des quatre configurations basées sur BTD n'est bonne ;

— l'ensemble *Indéterminé* contient le reste des instances difficiles.

Le tableau 6.7 nous montre comment les instances de chaque classe sont réparties entre ces ensembles. Beaucoup d'instances de *Décompose* viennent de la classe STRUCT qui contient des instances structurées (au sens de la décomposition arborescente). Ce n'est pas une surprise que les approches basées sur BTD ont de meilleures performances que les approches non basées sur BTD (voir par exemple [JT03b]). Comme BTD n'a jamais été comparé avec des mécanismes de retour arrière dynamiques, il est

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

	Nombre d'instances difficiles							Taille sép (moy)	Largeur (moy)
	ACAD	PATT	QRND	RAND	REAL	STRUCT	Total		
Décompose	7	8	1	14	5	177	212	4,7%	17,1%
Décompose pas	5	50	12	23	77	63	230	25,3%	31,8%
Indéterminé	9	35	5	99	3	29	180	32,9%	54,5%

Tableau 6.7 – *Description de trois ensembles d'instances. Pour chaque ensemble, le tableau montre le nombre d'instances difficiles dans chaque classe de l'ensemble d'instances, la taille d'un séparateur et la largeur arborescente moyenne (en pourcentage du nombre de variables) de la décomposition arborescente.*

intéressant de noter que BTD a de meilleures performances que des approches basées sur ces retours arrière dynamiques sur beaucoup de ces instances structurées. Seulement 35 instances de l'ensemble de problèmes XCSP appartiennent à l'ensemble *Décompose* : beaucoup d'instances de cet ensemble de problèmes XCSP ne possèdent pas de structure statique pouvant être exploitée par BTD. En regardant les paramètres de la décomposition arborescente, nous notons que les instances de *Décompose* ont une largeur arborescente plus petite (la moitié de celle de l'ensemble *Décompose pas*) et une taille de séparateur plus petite également (un cinquième de celle de l'ensemble *Décompose pas*). Les instances de l'ensemble *Indéterminé* ont une grande largeur arborescente. En fait, quand la largeur arborescente est proche du nombre de variables du problème, BTD se comporte comme CBT, car presque toutes les variables appartiennent au même cluster.

Sur certaines instances, le plus remarquablement sur les instances RLFAP de la classe REAL, la décomposition est bonne mais l'instance est dans l'ensemble *Décompose pas*. Ces instances sont faciles (la solution étant trouvée rapidement par (CBT,FC,d)) mais énormes (jusqu'à 900 variables). Restreindre la recherche à des clusters peut freiner considérablement la découverte d'une solution, car nous empêchons l'heuristique de choix de variable de nous guider rapidement vers la solution.

Nous faisons un focus sur les instances STRUCT (présentées dans le tableau 5.3), générées aléatoirement avec une structure, dans le tableau 6.8. Comme vu dans le tableau précédent, la majorité des instances de STRUCT sont dans la catégorie *Décompose*. Quelques classes sont néanmoins équilibrées entre *Décompose* et *Décompose pas* (les classes $c7, i7, c8, i8, c11, i11, c12, i12$). Dans ces classes, les instances de *Décompose pas* sont dans la grande majorité des cas mieux résolues par (CBT,MAC,w) – et cette configuration est souvent la seule meilleure–. Notons que l'ensemble *Indeterminé* est effectivement très peu représenté pour ces instances structurées.

6.3 Discussion

Nous avons détaillé les résultats de vingt-quatre configurations sur notre ensemble de 1092 instances, et ce avec quinze exécutions par combinaison de configuration et d'instance. Nous avons pu dégager des premières tendances et décider ainsi d'ignorer à la fois des instances trop simples à résoudre et des configurations trop peu intéressantes au vu des performances des autres configurations.

Une fois cette étape accomplie, il nous reste 622 instances non triviales et 13 configurations nécessaires pour couvrir ces instances. Nous avons pu étudier ces 13 configurations sous deux prismes différents. Tout d'abord nous avons inspecté individuellement chacune des configurations pour voir se dégager des tendances sur le type d'ins-

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

	Nombre d'instances difficiles			Taille sép	Largeur
	Décompose	Décompose pas	Indéterminé	(moy)	(moy)
c1,i1	13	0	2	3,3%	10%
c2,i2	15	2	2	3,3%	10%
c3,i3	11	2	2	3,3%	10%
c4,i4	7	3	2	3,3%	10%
c5,i5	14	4	2	2%	8%
c6,i6	14	4	1	2%	8%
c7,i7	8	8	2	2%	8%
c8,i8	10	7	2	2%	8%
c9,i9	14	4	1	2%	8%
c10,i10	12	4	2	2%	6%
c11,i11	7	7	3	2%	6%
c12,i12	8	6	3	2%	6%
c13,i13	16	2	2	2%	6%
c14,i14	18	1	1	1%	3%
c15,i15	10	8	2	1%	3%

Tableau 6.8 – *Description de trois ensembles d'instances. Pour chaque ensemble, le tableau montre le nombre d'instances difficiles pour chaque jeu de paramètres de la classe STRUCT, la taille d'un séparateur et la largeur arborescente moyenne (en pourcentage du nombre de variables) de la décomposition arborescente.*

tance qu'elles résolvaient le mieux. Ensuite nous avons tenté de déterminer si l'usage de la décomposition pouvait départager ces 13 configurations.

Comme les 13 configurations sont indispensables pour couvrir notre ensemble d'instances difficiles, nous voudrions savoir, étant donnée une instance particulière, quelle configuration utiliser ; c'est l'objet de la prochaine partie.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Troisième partie

Sélection automatique de configuration

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Chapitre 7

Sélection de solveur dans un portfolio

Sommaire

7.1 Cadre basique du sélecteur 83
7.1.1 Propriétés utilisées pour décrire une instance CSP 84
7.1.2 Entraînement 85
7.2 Choix d'un sous-ensemble de configurations 85
7.3 Discussion 89

Les résultats expérimentaux rapportés dans le chapitre précédent (chapitre 6) nous montrent que les meilleures configurations sur certaines instances sont médiocres sur d'autres instances, de telle sorte qu'elles sont loin d'avoir les meilleurs taux de succès moyen sur l'ensemble de problèmes. Cette illustration du théorème *no free lunch* motive notre étude d'un sélecteur d'algorithme par instance. Ce sélecteur vise à sélectionner une bonne configuration pour chaque nouvelle instance à résoudre.

Dans cette étude, nous ne visons pas à améliorer l'état de l'art des sélecteurs d'algorithmes par instance présentés au chapitre 3. Nous nous focalisons sur un point clé de ces approches : la sélection des solveurs à inclure dans le portfolio. En effet, [AGM13] nous montre que de meilleures performances peuvent être obtenues avec des portfolios plus restreints. Ceci est dû au fait que l'apprentissage effectué pour choisir une configuration est plus complexe lorsqu'il y a plus de configurations à départager. Cependant il est aussi important que le portfolio contienne un nombre suffisant de solveurs pour assurer la présence d'un bon solveur pour chaque instance.

Dans ce chapitre nous présentons tout d'abord le cadre général du sélecteur de configurations en section 7.1, puis nous introduisons deux stratégies pour choisir les configurations à inclure dans le portfolio en section 7.2. Ces deux stratégies seront évaluées expérimentalement au chapitre 8.

7.1 Cadre basique du sélecteur

Nous considérons un cadre classique similaire au cadre de [AGM13]. L'idée est d'entraîner un classifieur supervisé (comme vu en section 3.1.2) en lui donnant un ensemble d'instances CSP étiquetées : chaque instance de l'ensemble d'entraînement est décrite par un vecteur de propriétés et est associée avec une étiquette correspondant à la meilleure configuration du portfolio pour cette instance. Cette étape d'entraînement permet au classifieur d'apprendre un modèle de sélection. Ensuite, ce modèle est

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

utilisé pour sélectionner une configuration pour une nouvelle instance, en se basant seulement sur son vecteur de propriétés.

7.1.1 Propriétés utilisées pour décrire une instance CSP

Le choix des propriétés décrivant les instances est primordial. En effet, dans une tâche de classification, le pré-traitement des données est certainement la partie la plus importante du travail. Sans données préparées consciencieusement, même le meilleur algorithme de classification aura de piètres performances. Nous présentons ici les propriétés retenues dans notre étude.

Chaque instance est décrite par un vecteur de propriétés. Nous considérons des propriétés classiques, similaires à celles utilisées dans [OHH+08, AGM13] par exemple. La différence principale est que nous profitons des informations extraites de la décomposition arborescente (voir la section 2.2.4) pour ajouter quelques propriétés, puisque le tableau 6.7 nous montre que les performances de BTD dépendent de la largeur arborescente et de la taille des séparateurs. Nous espérons ainsi permettre au classifieur de mieux déterminer quand utiliser une configuration exploitant une décomposition arborescente.

Souvenons-nous que le graphe des contraintes d'une instance peut contenir plusieurs composantes connexes. Nous pouvons donc ici aussi glaner quelques propriétés supplémentaires.

Plus précisément, nous extrayons les propriétés statiques suivantes pour chaque instance :

— nombre de variables, nombre de contraintes, taille des domaines (moyenne et écart-type),
— dureté des contraintes (ratio de tuples interdits sur tous les tuples possibles, moyenne et écart-type),
— degrés des variables dans le graphe des contraintes (moyenne et écart-type),
— nombre de composantes connexes dans le graphe des contraintes (moyenne et écart-type), nombre de variables dans une composante connexe (moyenne et écart-type), nombre de contraintes dans une composante connexe (moyenne et écart-type),
— nombre de clusters dans la décomposition arborescente, taille maximum d'un cluster, taille maximum d'un séparateur, densité des contraintes dans un cluster (moyenne et écart-type).

Afin de récupérer plus d'informations sur les instances à résoudre, nous exécutons pour une courte période une configuration sur l'instance à résoudre. Nous en retirons des propriétés dynamiques. L'exécution est limitée à une seconde.

Comme (CBT,MAC,w) est la meilleure configuration pour cette limite de temps, c'est (CBT,MAC,w) que nous lançons pour récupérer ces informations. De plus, cette configuration nous permet de récupérer les poids des variables utilisés par l'heuristique wdeg ainsi que le nombre de valeurs filtrées par MAC. Nous récupérons les propriétés dynamiques suivantes :

— nombre de nœuds dans l'arbre de recherche, profondeur maximum d'un nœud dans l'arbre de recherche, nombre d'échecs,
— nombre de valeurs filtrées par MAC (moyenne et écart-type),
— poids d'une variable (moyenne et écart-type).

Pour obtenir des informations sur la progression de la résolution, nous récupérons ces informations dynamiques à trois temps différents dans la résolution : à $0,25$, à $0,5$ et

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

à 1 seconde. Nous espérons ainsi départager différents comportements au début de la résolution.

7.1.2 Entraînement

Maintenant que nos instances sont décrites par leur vecteur de propriétés, nous pouvons présenter le protocole d'entraînement de notre sélecteur de configuration.

Étant donnés :

— un portfolio de configurations P,
— un ensemble d'entraînement I d'instances tel que chaque instance $i \in I$ soit décrite par un vecteur de propriétés,
— une fonction de supervision $s : I \to P$ associant à chaque instance $i \in I$ la configuration $s(i) \in P$ qui est la meilleure pour résoudre i,

le but est d'entraîner un classifieur associant des instances avec leurs meilleures configurations. Nous présentons ici la démarche ainsi que les outils utilisés.

Dans cette étude, nous nous sommes servis de la bibliothèque Weka [HDW94,HFH+09] pour cette tâche. Nous avons comparé différents classifieurs supervisés implémentés dans Weka. Les meilleurs résultats de classification sont obtenus par *ClassificationViaRegression* avec les paramètres par défaut [FWI+98] ; nous utilisons donc ce classifieur dans nos expérimentations.

Une fois le classifieur entraîné sur l'ensemble d'entraînement, nous pouvons l'utiliser pour sélectionner dynamiquement la meilleure configuration de notre solveur générique pour chaque nouvelle instance à résoudre. Plus précisément, pour résoudre une nouvelle instance i nous procédons comme suit. D'abord nous exécutons (CBT,MAC,w) sur i avec une limite de temps CPU de une seconde ; si i n'est pas résolue, nous en extrayons ses propriétés statiques et dynamique (basées sur l'exécution de (CBT,MAC,w)) ; nous donnons ces propriétés au classifieur qui renvoie une configuration et nous exécutons cette configuration sur i. Notons que le temps passé à extraire les propriétés et classifier i est très court : moins de $0,1$ secondes en moyenne. Ce temps sera donc négligeable comparé au temps total de résolution d'une instance.

7.2 Choix d'un sous-ensemble de configurations

Nos instances étant décrites par un vecteur de propriétés et le sélecteur étant prêt à être entraîné, il ne reste plus qu'à choisir un ensemble de configurations constituant notre portfolio. C'est sur cette étape que nous proposons une nouvelle stratégie de sélection de portfolio.

Un point clef de la sélection d'algorithme par instance est de choisir les configurations inclues dans le portfolio. Le but est idéalement de choisir des configurations avec des comportements complémentaires afin que le portfolio contienne une bonne configuration pour chaque instance. Nous pouvons inclure dans notre portfolio les treize configurations non dominées identifiées dans le chapitre 7. Cependant, plus le portfolio est grand, plus l'apprentissage d'un modèle de classification est difficile. Ainsi, de meilleurs résultats peuvent être obtenus avec moins de configurations, comme décrit dans [AGM13].

De façon plus précise, il s'agit de choisir un sous-ensemble S_k de k configurations (où $k \in [2; 13]$ est un paramètre à fixer), étant donné un ensemble I d'instances d'entraînement. La qualité d'un portfolio peut être évaluée par rapport aux performances d'un

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

meilleur configurateur virtuel (VBC) : étant donné un sous-ensemble S_k de k configurations, un VBC choisit toujours la meilleure configuration de S_k pour chaque $i \in I$.

Nous décrivons et comparons deux stratégies pour choisir le sous-ensemble S_k : la stratégie utilisée dans [AGM13] et une nouvelle stratégie visant à sélectionner une configuration *bonne* pour l'instance à résoudre, plutôt que de viser à trouver une configuration qui ne fait « que » résoudre ce type d'instance.

Solved

La première stratégie, appelée *Solved*, est celle utilisée dans [AGM13]. Cette stratégie choisit pour S_k les k configurations qui maximisent le nombre d'instances résolues par un VBC à la limite de temps CPU donnée. Les égalités sont départagées en minimisant le temps de résolution du VBC en moyenne sur les instances résolues de I.

La constitution d'un portfolio selon cette stratégie dépend très fortement de la limite de temps CPU fixée. Plus cette limite est importante et plus la stratégie va choisir des configurations « robustes », capables de résoudre un plus grand nombre d'instances, quitte à ce que ces configurations soient plus lentes pour résoudre des instances plus faciles.

Good

La seconde stratégie, appelée *Good*, est notre proposition. Cette stratégie choisit pour S_k les k configurations maximisant le nombre d'instances pour lesquelles S_k contient une bonne configuration. Rappelons qu'une bonne configuration est une configuration qui n'est pas statistiquement différente de la meilleure configuration pour cette instance. Les égalités sont départagées en maximisant le nombre d'instances de I résolues par un VBC à une limite de temps CPU donnée.

Cette seconde stratégie est également dépendante de la limite de temps CPU fixée, mais ce critère devient secondaire et le premier critère vise à constituer un portfolio contenant une configuration adaptée à chaque instance.

Algorithme de sélection

Pour les deux stratégies, trouver le sous-ensemble optimal S_k est NP-difficile : c'est un problème de couverture d'ensemble entre configurations et instances, où une configuration s couvre une instance i si s peut résoudre i dans la limite de temps CPU (pour *Solved*) ou si s est bonne pour i (pour *Good*). Dans cette étude nous approximons la solution à ce problème de façon gloutonne : en partant du sous-ensemble S_1 contenant la configuration couvrant le plus d'instances, nous définissons S_i en ajoutant à S_{i-1} la configuration qui augmente le plus le nombre d'instances couvertes. Cette méthode est valable pour les deux stratégies.

Comparaison des portfolios

En considérant les quinze exécutions de nos treize configurations sur les 622 instances difficiles, nous obtenons les ordres de sélection suivants pour les deux stratégies considérées dans notre étude :

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

	Solved	*Good*
1	(CBT,MAC,w)	(BTD,FC,d)
2	(BTD,FC,w)	(CBT,MAC,w)
3	(CBJR,FC,w)	(BTD,FC,w)
4	(DR,FC,w)	(CBT,FC,d)
5	(CBJ,MAC,w)	(CBT,MAC,d)
6	(CBT,MAC,d)	(DR,FC,w)
7	(BTD,FC,d)	(CBJ,MAC,w)
8	(BTD,MAC,w)	(BTD,MAC,w)
9	(CBT,FC,d)	(CBJ,FC,d)
10	(CBJ,FC,d)	(CBJR,FC,w)
11	(CBJ,MAC,d)	(CBT,FC,w)
12	(BTD,MAC,d)	(BTD,MAC,d)
13	(CBT,FC,w)	(CBJ,MAC,d)

Bien sur, cet ordre dépend fortement de la composition de l'ensemble d'instances. Par exemple, si nous enlevons la moitié des instances STRUCT, la stratégie *Good* sélectionne (CBT,FC,d) en troisième position et (BTD,FC,w) en quatrième position, le reste étant inchangé. Cependant, si nous enlevons toutes les instances STRUCT, l'ordre devient très différent et la meilleure configuration basée sur BTD devient (BTD,FC,w) et est sélectionnée en quatrième position.

Voici l'ordre de sélection donné par la stratégie *Good* avec la moitié des instances STRUCT en moins :
1 (BTD,FC,d) ; 2 (CBT,MAC,w) ; 3 (CBT,FC,d) ; 4 (BTD,FC,w) ; 5 (CBT,MAC,d) ; 6 (DR,FC,w) ; 7 (CBJ,MAC,w) ; 8 (BTD,MAC,w) ; 9 (CBJ,FC,d) ; 10 (CBJR,FC,w) ; 11 (CBT,FC,w) ; 12 (CBJ,MAC,d) ; 13 (BTD,MAC,d).

Et si nous enlevons toutes les instances STRUCT, la stratégie *Good* donne l'ordre suivant :
1 (CBT,FC,d) ; 2 (CBT,MAC,w) ; 3 (CBT,MAC,d) ; 4 (BTD,FC,w) ; 5 (DR,FC,w) ; 6 (CBJ,MAC,w) ; 7 (CBJ,FC,d) ; 8 (CBJR,FC,w) ; 9 (CBT,FC,w) ; 10 (BTD,MAC,w).

Il est intéressant de constater que comme (CBJ,MAC,d), (BTD,MAC,d) et (BTD,FC,d) ne sont l'unique bonne configuration que sur des instances STRUCT (voir le tableau 6.6), nous n'avons pas besoin de ces trois configurations pour couvrir notre ensemble d'instances sans les instances STRUCT.

Pour la stratégie *Solved* il est pertinent de comparer l'ordre de sélection quand le temps limite imparti est différent.

Voici l'ordre proposé par la stratégie *Solved* avec un temps limite de 100 secondes :
1 (CBT,MAC,w) ; 2 (BTD,FC,d) ; 3 (DR,FC,w) ; 4 (BTD,FC,w) ; 5 (CBJR,FC,w) ; 6 (CBT,FC,d) ; 7 (CBJ,MAC,w) ; 8 (CBJ,FC,d) ; 9 (CBJ,MAC,d) ; 10 (CBT,FC,w) ; 11 (BTD,MAC,w).

Et si le temps limite est de 500 secondes, l'ordre pour la stratégie *Solved* devient :
1 (CBT,MAC,w) ; 2 (BTD,FC,d) ; 3 (CBJR,FC,w) ; 4 (BTD,FC,w) ; 5 (DR,FC,w) ; 6 (CBJ,MAC,w) ; 7 (CBT,FC,d) ; 8 (BTD,MAC,d) ; 9 (BTD,MAC,w).

La première configuration choisie reste la même que pour la limite de temps à 1800 secondes. Par contre la configuration (BTD,FC,w) est délogée pour se retrouver en quatrième position avec le temps limite à 100 ou 500 secondes. C'est la configuration (BTD,FC,d) qui prend sa place, peut-être que la simplicité de ddeg permet de résoudre plus rapide-

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

k	1	2	3	4	5	6	7	8	9	10	11	12	13
solved	22.3	45.7	47.6	50.5	52.4	59.6	86.0	88.4	96.8	98.7	98.9	99.2	100
good	44.7	62.5	72.5	81.3	88.6	91.5	93.4	95.7	97.6	98.7	99.5	99.8	100

Tableau 7.1 – *Comparaison de* Solved *et* Good *en pourcentage de bonnes configurations pour l'ensemble des instances difficiles*

ment (en dessous de 500 secondes) plus d'instances que wdeg.

Le tableau 7.1 présente le pourcentage de bonnes configurations (pour les instances difficiles) présentes dans les portfolios choisis selon les stratégies *Solved* et *Good*. Nous pouvons vérifier que la stratégie *Good* choisit effectivement en priorité les configurations ajoutant le plus d'instances couvertes, c'est-à-dire les instances pour lesquelles une au moins des configurations présentes dans le portfolio est bonne pour cette instance. Il est intéressant de noter que la stratégie *Good* couvre 80% d'instances avec quatre configurations seulement, là où la stratégie *Solved* requiert l'ajout de la septième configuration pour atteindre ce palier. Pour la stratégie *Solved*, le fait de chercher les configurations résolvant le plus d'instances à la limite de temps allouée ne désigne donc apparemment pas les configurations bonnes pour le plus d'instances possible. Comparons maintenant les deux stratégies sur le potentiel de résolution qu'elles offrent à différentes limites de temps.

Notons S_k^s le sous-ensemble qui contient chaque configuration dont le rang est en dessous de ou égal à k pour la stratégie $s \in \{Solved, Good\}$, et VBC(S_k^s) le VBC associé à S_k^s.

Le tableau 7.2 compare les deux stratégies en terme de taux de succès des VBC associés. Notons que VBC(S_k^{Solved})=VBC(S_k^{Good}) quand $k = 10$ ou $k = 13$ car les ensembles S_k^{Solved} et S_k^{Good} sont les mêmes dans ces deux cas. Aussi, VBC(S_k^{Solved}) fait mieux que VBC(S_k^{Good}) à 1800 secondes de temps limite quand $k \leq 9$ et les deux approches sont équivalentes quand $k \geq 10$. Cela vient du fait que la stratégie *Solved* maximise le nombre d'instance résolues à la limite de temps. En contrepartie, VBC(S_k^{Good}) fait mieux que VBC(S_k^{Solved}) pour des temps limites plus petits ou des valeurs de k moindres. Cela vient du fait que la stratégie *Good* maximise le nombre d'instances pour lesquelles le portfolio contient une bonne configuration, indépendamment de la limite de temps. Ces observations sont cohérentes avec le but de chaque stratégie.

Par exemple, quand $k = 4$, la différence entre les taux de succès de VBC(S_4^{Good}) et VBC(S_4^{Solved}) est égale à $2,2$, puis $3,5$, puis $2,4$, puis $1,3$, et enfin $0,2$ quand la limite de temps est égale à 1, puis 5, puis 10, puis 50 et enfin 100 secondes respectivement, tandis que cette différence devient négative après 100 secondes. Cependant, la différence est moins importante ($-0,4$, puis $-0,7$ et enfin $-0,7$ à 500, puis 1000 et enfin 1800 secondes respectivement). En fait, avec S_3^{Solved}={(CBT,MAC,w), (BTD,FC,w), (CBJR,FC,w)}, un VBC est capable de résoudre 99.4% des exécutions, mais S_3^{Solved} ne contient une bonne configuration que pour 294 des 622 instances difficiles. En ajoutant de nouvelles configurations à S_3^{Solved}, nous n'améliorons que très peu le taux de succès du VBC correspondant. La configuration qui augmente le plus le nombre d'instances résolues est (DR,FC,w) et elle nous permet de résoudre 26 exécutions de plus (sur 622 fois quinze exécutions). Cependant, (DR,FC,w) est une bonne configuration pour un nombre restreint d'instances et l'ajouter à S_3^{Solved} augmente le nombre d'instances difficiles pour lesquelles nous avons une bonne configuration de 24. En comparaison, ajouter (BTD,FC,d) à S_3^{Solved} nous permettrait de résoudre 12 exécutions de plus (au lieu de 26, sur 622 fois quinze) mais cela augmenterait le nombre d'instances difficiles pour

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

	1	5	10	50	100	500	1000	1800
Solved								
2	52.0	68.9	75.4	87.5	91.8	96.7	98.2	98.7
3	52.4	69.2	75.9	87.8	92.5	97.6	99.0	99.4
4	52.5	69.4	76.1	88.2	92.8	97.8	99.1	99.5
5	52.7	69.4	76.1	88.3	93.0	97.8	99.2	99.6
6	54.5	69.7	76.4	88.8	93.1	97.8	99.2	99.7
7	55.6	72.3	78.7	90.2	94.0	98.3	99.3	99.7
8	55.7	72.3	78.8	90.4	94.0	98.3	99.4	99.7
9	56.2	73.4	79.3	90.5	94.1	98.4	99.4	99.7
10	56.5	73.5	79.5	90.7	94.2	98.4	99.4	99.7
11	56.5	73.5	79.5	90.8	94.3	98.4	99.4	99.7
12	56.5	73.5	79.5	90.8	94.3	98.4	99.4	99.7
13	56.6	73.7	79.6	90.8	94.3	98.4	99.4	99.7
Good								
2	52.7	70.6	77.2	88.7	92.3	96.9	98.2	98.6
3	53.2	71.6	77.8	89.0	92.8	97.3	98.4	98.8
4	54.7	72.9	78.5	89.5	93.0	97.4	98.4	98.8
5	55.8	73.1	78.7	89.7	93.0	97.4	98.4	98.8
6	55.9	73.3	79.0	90.3	93.6	97.8	98.8	99.3
7	56.0	73.3	79.0	90.3	93.9	98.1	99.2	99.6
8	56.1	73.3	79.2	90.5	93.9	98.1	99.2	99.6
9	56.4	73.5	79.3	90.6	94.1	98.2	99.2	99.6
10	56.5	73.5	79.5	90.7	94.2	98.4	99.4	99.7
11	56.5	73.7	79.6	90.7	94.2	98.4	99.4	99.7
12	56.5	73.7	79.6	90.7	94.2	98.4	99.4	99.7
13	56.6	73.7	79.6	90.8	94.3	98.4	99.4	99.7

Tableau 7.2 – *Comparaison de* Solved *et* Good. *Chaque ligne donne à la suite : le nombre k de configurations choisies dans S_k et pour chaque limite de temps en secondes le pourcentage d'exécutions réussies d'un VBC construit sur les ensembles définis par* Solved *et* Good *(sur 15 exécutions sur les 1092 instances). Pour chaque couple (S_k,temps), nous surlignons la stratégie avec le plus haut taux de succès.*

lesquelles nous avons une bonne configuration de 168 (au lieu de 24, sur 622 instances).

Nous avons maintenant les performances théoriques de sélecteurs basées sur des portfolio choisis selon deux stratégies différentes.

7.3 Discussion

Nous avons accès à des propriétés permettant de décrire nos instances de façon pertinente. Grâce à cela nous pouvons entraîner un classifieur selon le protocole qui a été décrit en section 7.1.2.

Nous avons décrit deux stratégies pour choisir le portfolio de configurations de notre sélecteur. Ces deux stratégies ont des buts différents qui sont visibles au travers de VBC utilisant ces stratégies. Nous avons également constaté l'influence de l'ensemble de problèmes sur la constitution des portfolios.

Après l'étude de ces résultats théoriques, nous pouvons passer à l'expérimentation en pratique de ces différents sélecteurs de configurations.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Chapitre 8

Évaluation expérimentale

Sommaire

8.1 Protocole expérimental 91
8.2 Qualité des configurations sélectionnées 92
8.3 Comparaison des taux de succès 93
8.4 Discussion 95

Les sélecteurs de configurations sont désormais fonctionnels : nous disposons de différents ensembles de configurations pour comparer plusieurs sélecteurs, nous avons un ensemble d'instances sur lequel tester ces sélecteurs et une représentation de ces instances sous formes de vecteur de propriété.

Dans ce chapitre nous allons décrire le protocole expérimental utilisé pour mener la comparaison des différents sélecteurs en section 8.1, puis nous analyserons leurs résultats. Pour cela nous pourrons étudier tout d'abord la qualité des configurations sélectionnées par rapport aux autres configurations du sélecteur en section 8.2 et ensuite les taux de succès à différentes limites de temps des sélecteurs en section 8.3.

8.1 Protocole expérimental

Nous considérons les 1092 instances de l'ensemble d'instances présenté au chapitre 5. L'ensemble d'entraînement est composé des 622 instances non triviales de cet ensemble de problèmes (voir la section 6.2.3).

Nous procédons en *leave-one-out* : pour chaque instance i de l'ensemble d'instances, si i est une instance non triviale appartenant à l'ensemble d'entraînement nous enlevons i de cet ensemble d'entraînement et nous entraînons (voir section 7.1.2) le classifieur sur toutes les instances sauf i ; enfin nous demandons au classifieur de choisir une configuration pour i.

Nous garantissons ainsi la validité du protocole car nous n'utilisons pas les résultats connus d'une instance pour classer cette même instance. Ceci est répété pour chaque sélecteur présenté en section 7.2.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

classé comme	a	b	c	d	e	f	g	h	i	j	k
a = (CBT,FC,d)	96	4	2	5	0	0	0	1	25	1	0
b = (CBT,FC,w)	2	20	3	3	0	0	0	1	0	0	3
c = (CBT,MAC,d)	3	1	36	6	0	0	0	0	11	0	2
d = (CBT,MAC,w)	4	0	9	34	0	0	0	3	33	6	5
e = (DBT,FC,d)	0	0	0	0	0	0	0	0	0	0	0
f = (DBT,MAC,d)	0	0	0	0	0	0	0	0	0	0	0
g = (CBJ,MAC,w)	0	0	0	0	0	0	0	0	0	0	0
h = (CBJR,FC,w)	3	1	2	4	0	0	0	2	11	3	5
i = (BTD,FC,d)	34	0	0	4	0	0	0	1	130	4	1
j = (BTD,FC,w)	4	1	0	0	0	0	0	2	49	5	3
k = (BTD,MAC,w)	0	3	1	7	0	0	0	1	13	2	7

Tableau 8.1 – *Matrice de confusion pour S_{11}^{Solved}. La ligne donne la bonne classe et la colonne la classe donnée par le classifieur.*

8.2 Qualité des configurations sélectionnées

La configuration apprise d'une instance i est la configuration renvoyée par le classifieur. Nous donnons un exemple de matrice de confusion dans le tableau 8.1 pour le sélecteur S_{11}^{Solved}.

Nous disons que i est *bien classée* si sa configuration sélectionnée est la meilleure pour i parmi l'ensemble S_k^s de configurations candidates (ou si elle n'est pas statistiquement différente de la meilleure configuration pour i dans S_k^s). La deuxième colonne de la partie gauche du tableau 8.2 donne le pourcentage d'instances non triviales bien classées pour *Solved* et puis pour *Good*. Cela nous montre que ce pourcentage décroît quand le nombre k de configurations dans S_k^s augmente, à la fois pour *Solved* et *Good* : ce pourcentage décroît de $81,7\%$ et $84,6\%$ avec deux configurations à $65,4\%$ pour les deux avec treize configurations. Cependant, la partie gauche du tableau 8.2 montre aussi que les configurations apprises pour les instances mal classées sont souvent des configurations qui ont de bons résultats : étant donné un ensemble S_k de configurations et étant donnée une instance i, nous notons chaque configuration de 1 à k selon ses performances sur i (la configuration classée n°1 étant la meilleure pour i et la configuration n°k étant la plus mauvaise). Par exemple, regardons les résultats pour S_{13} : pour $65,4\%$ des instances, la configuration apprise est la meilleure ou n'est pas significativement moins bonne que la meilleure ; pour $9,8\%$ des instances, c'est la seconde meilleure ou une configuration qui n'est pas significativement moins bonne que la seconde meilleure ; … ; et enfin, pour $8,9\%$ des instances c'est la septième meilleure ou pire.

Le fait que la configuration apprise soit bien classée pour une instance i n'implique pas forcément qu'elle soit bonne pour i (sauf quand $k = 13$) : cela dépend si S_k contient une bonne configuration pour i ou non. La partie droite du tableau 8.2 affiche le pourcentage d'instances non triviales pour lesquelles la configuration apprise est bonne. Pour la stratégie *Solved* ce pourcentage croit de $36,7\%$ avec S_2^{Solved} à $65,8\%$ avec S_{11}^{Solved}, tandis que pour la stratégie *Good*, ce pourcentage croit de $54,5\%$ avec S_2^{Good} à 69% avec S_8^{Good}.

Ce pourcentage de bonnes configurations sélectionnées nous montre que la stratégie *Good* remplit bien son objectif. En effet, quel que que soit le nombre de configurations dans le sélecteur, de 2 à 13, cette stratégie choisit plus de bonnes configurations.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

	Classement des configurations apprises							% bonnes conf.
k	1	2	3	4	5	6	≥7	
				Solved				
2	81.7	18.3						36.7
3	78.6	12.4	9.0					36.5
4	78.6	10.6	6.3	4.5				37.3
5	77.0	7.2	6.6	4.7	4.5			38.4
6	73.2	6.6	6.6	3.9	5.8	4.0		41.3
7	66.1	13.8	5.5	4.3	5.5	2.4	2.4	59.8
8	63.8	11.6	5.9	5.6	5.9	3.1	4.0	58.7
9	65.6	11.4	6.1	4.3	4.0	3.4	5.1	64.8
10	66.1	10.9	5.9	4.0	3.2	3.5	6.3	65.4
11	66.2	10.0	6.3	3.5	3.4	1.9	8.6	65.8
12	64.8	9.8	6.1	3.9	4.3	1.9	9.1	64.1
13	65.4	9.8	6.8	3.4	3.9	1.8	8.9	65.4
				Good				
2	84.6	15.4						54.5
3	77.3	16.2	6.4					56.4
4	75.7	15.8	6.8	1.8				61.4
5	75.1	14.6	5.9	3.2	1.1			67.2
6	71.5	13.8	7.1	5.0	2.3	0.3		66.4
7	71.2	11.9	5.5	3.9	4.7	2.6	0.3	67.8
8	70.7	10.5	5.8	5.8	3.1	1.9	2.3	69.0
9	67.7	9.8	5.6	7.1	3.9	2.4	3.5	67.2
10	66.1	10.9	5.9	4.0	3.2	3.5	6.3	65.4
11	66.7	10.5	6.1	3.9	2.7	2.9	7.2	66.6
12	65.8	10.5	6.8	3.7	3.2	2.1	8.1	65.6
13	65.4	9.8	6.8	3.4	3.9	1.8	8.9	65.4

Tableau 8.2 – *Classement des configurations apprises. Pour chaque ensemble S_k et chaque rang $j \in \{1,\ldots,k\}$, le tableau de gauche affiche le pourcentage d'instances difficiles dont la configuration apprise est la j^e meilleure parmi les k configurations dans S_k. Pour chaque ensemble S_k, le tableau de droite donne le pourcentage d'instances non triviales pour lesquelles la configuration apprise est une bonne configuration.*

L'écart est toujours au moins de 10 points pour k compris entre 2 et 8. Cet écart est maximum pour $k = 5$ et atteint alors $28,8$ points.

Nous constatons que pour k valant de 3 à 6, les sélecteurs utilisant la stratégie *Solved* classent un peu mieux les instances que les sélecteurs utilisant la stratégie *Good*. Pour le reste des k c'est l'inverse. Il ne semble pas que la stratégie de sélection favorise significativement les taux de classification. En tout cas, cela n'empêche pas la stratégie *Good* de fournir des sélecteurs qui choisissent bien plus souvent de bonnes configurations pour les instances.

8.3 Comparaison des taux de succès

Le tableau 8.3 donne le pourcentage d'instances résolues à différentes limites de temps pour la meilleure configuration, (CBT,MAC,w), et pour le sélecteur d'algorithme par instance avec différents portfolios S_k^s avec $k \in [2;13]$ et $s \in \{Solved, Good\}$ en

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

	1	5	10	50	100	500	1000	1800
Taux de succès de (CBT,MAC,w) (moyenne sur 15 essais) :								
	47.1	61.5	68.3	80.5	85.2	92.3	94.3	95.4
Solved								
2	47.1	64.2	72.1	84.7	88.8	94.4	**96.0**	**96.6**
3	47.1	64.5	71.9	84.4	88.6	94.3	95.6	96.1
4	47.1	64.8	72.1	84.7	88.7	94.4	95.7	96.2
5	47.1	64.9	72.0	84.5	88.6	94.4	95.8	96.3
6	47.1	64.3	71.5	83.5	87.7	93.2	94.6	95.2
7	47.1	66.7	73.1	85.0	88.4	94.0	95.0	95.7
8	47.1	66.0	72.7	84.8	88.0	93.7	94.8	95.6
9	47.1	67.8	74.0	85.1	88.4	94.0	94.9	95.5
10	47.1	67.9	73.9	85.3	88.9	94.3	95.2	95.7
11	47.1	67.9	73.9	85.3	89.1	94.4	95.3	95.7
12	47.1	67.8	73.7	85.3	88.6	94.5	95.4	95.8
13	47.1	**68.5**	**74.5**	**85.8**	**89.2**	**94.6**	95.7	96.3
Good								
2	47.1	66.7	73.9	86.5	**90.2**	95.1	96.3	96.8
3	47.1	66.5	73.8	86.0	90.1	95.3	96.2	96.6
4	47.1	67.8	74.6	86.3	89.9	95.4	96.3	96.8
5	47.1	68.4	**75.1**	86.3	89.9	**95.6**	**96.5**	**96.9**
6	47.1	**68.7**	75.0	**86.6**	89.7	95.1	95.9	96.4
7	47.1	68.2	74.5	86.1	89.4	94.8	95.9	96.5
8	47.1	68.2	74.7	86.2	89.7	95.0	95.7	96.3
9	47.1	68.2	74.6	86.0	89.4	94.7	95.5	96.1
10	47.1	67.9	73.9	85.3	88.9	94.3	95.2	95.7
11	47.1	68.2	74.3	85.5	88.8	94.5	95.3	95.7
12	47.1	68.1	74.4	85.8	89.2	94.7	95.7	96.2
13	47.1	68.5	74.5	85.8	89.2	94.6	95.7	96.3

Tableau 8.3 – *Chaque ligne contient la taille k du portfolio suivie par le taux de succès du sélecteur d'algorithme par instance construit avec S_k^{Solved} et S_k^{Good} à différentes limites de temps (pour quinze exécutions sur 1092 instances). Pour chaque limite de temps et chaque taille k nous colorions en bleu la cellule avec le meilleur résultat s'il est significativement meilleur. Pour chaque limite de temps et chaque stratégie $s \in \{Solved, Good\}$ nous surlignons en gras le taux de succès le plus élevé quelle que soit la taille k. La première ligne du tableau rappelle le taux de succès de (CBT,MAC,w).*

moyenne sur quinze exécutions. Nous avons utilisé le test de Student avec $p = 0,01$ pour décider si les quinze taux de succès à un temps t et une taille k sont significativement différents pour les deux stratégies et nous colorions en bleu les meilleures stratégies quand le test est positif.

À une seconde, toutes les variantes du sélecteur ont les mêmes taux de succès que (CBT,MAC,w) puisque les sélecteurs lancent (CBT,MAC,w) pendant une seconde avant de commencer le processus de sélection. Après cinq secondes, toutes les variantes du sélecteur ont de meilleurs taux de succès que (CBT,MAC,w). La stratégie *Good* est significativement meilleure que la stratégie *Solved* quand $k \leq 9$, pour toute limite de temps. Quand $k \geq 10$, les deux stratégies ont souvent des résultats qui ne sont pas significativement différents.

Pour la stratégie *Solved*, les meilleurs résultats sont obtenus avec le plus grand port-

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

folio, S_{13}, jusqu'à 500 secondes. Après, les meilleurs résultats sont obtenus avec S_2. Pour la stratégie *Good*, les meilleurs résultats sont souvent obtenus avec un portfolio de cinq ou six configurations.

Les résultats sont de façon impressionnante en faveur de la stratégie *Good*. Le seul sélecteur de la stratégie *Solved* ayant un meilleur pourcentage de résolution aux temps présentés dans le tableau 8.3 est celui avec onze configurations à 100 secondes ; ce résultat n'est pas significativement meilleur que celui du sélecteur correspondant pour la stratégie *Good*.

Tous les sélecteurs composés de neuf configurations ou moins, utilisant la stratégie *Good*, ont des résultats significativement meilleurs que ceux utilisant la stratégie *Solved* et ce à chaque intervalle de temps présenté. Il semble néanmoins étonnant que les sélecteurs utilisant la stratégie *Solved* n'obtiennent pas de meilleurs résultats à 1800 secondes, qui est l'objectif visé de cette stratégie.

La figure 8.1 dessine l'évolution du pourcentage d'instances résolues avec une limite de temps CPU pour la meilleure configuration (CBT,MAC,w) et pour les sélecteurs S_5^{Good} et S_{13}^{Solved}. Cette figure donne aussi les résultats de VBC(S_5^{Good}) et VBC(S_{13}^{Solved}).

Cette figure illustre le tableau précédent. Elle illustre également la qualité des VBS présentés au chapitre précédent dans le tableau 7.2. Nous voyons ainsi que le VBS avec cinq configurations de la stratégie *Good* est très proche du VBC avec treize configurations. Par ailleurs, le gain à 1800 secondes de nos sélecteurs semble faible par rapport à la performance de (CBT,MAC,w). C'est entre 5 et 100 secondes que nos sélecteurs se démarquent fortement et permettent de résoudre plus d'instances. Enfin le sélecteur avec cinq configurations choisies par la stratégie *Good* est légèrement meilleur que le sélecteur à treize configurations (représentant un bon choix pour la stratégie *Solved*), notamment après 10 secondes.

8.4 Discussion

Dans ce chapitre nous avons poursuivi la comparaison des deux stratégies de sélection de configurations pour nos sélecteurs, *Solved* et *Good*. Les sélecteurs utilisant notre proposition, la stratégie *Good*, sélectionnent plus souvent de bonnes configurations pour résoudre les instances que les sélecteurs utilisant la stratégie *Solved*. L'objectif de base est donc atteint : plutôt que de maximiser le nombre d'instances résolues au temps limite, nous voulions résoudre « bien » un maximum d'instances – et cette partie est acquise.

Ce qui est intéressant est le fait que, ce faisant, les sélecteurs utilisant des configurations choisies via la stratégie *Good* résolvent plus d'instances, à chaque temps donné dans le tableau 8.3, que les sélecteurs basés sur la stratégie *Solved* et souvent de manière statistiquement significative. Il semble donc que l'utilisation de tests statistiques couplés à l'objectif de chercher les *bonnes configurations* engendre de meilleurs résultats que le fait de vouloir maximiser le nombre d'instances résolues à une limite de temps donnée.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

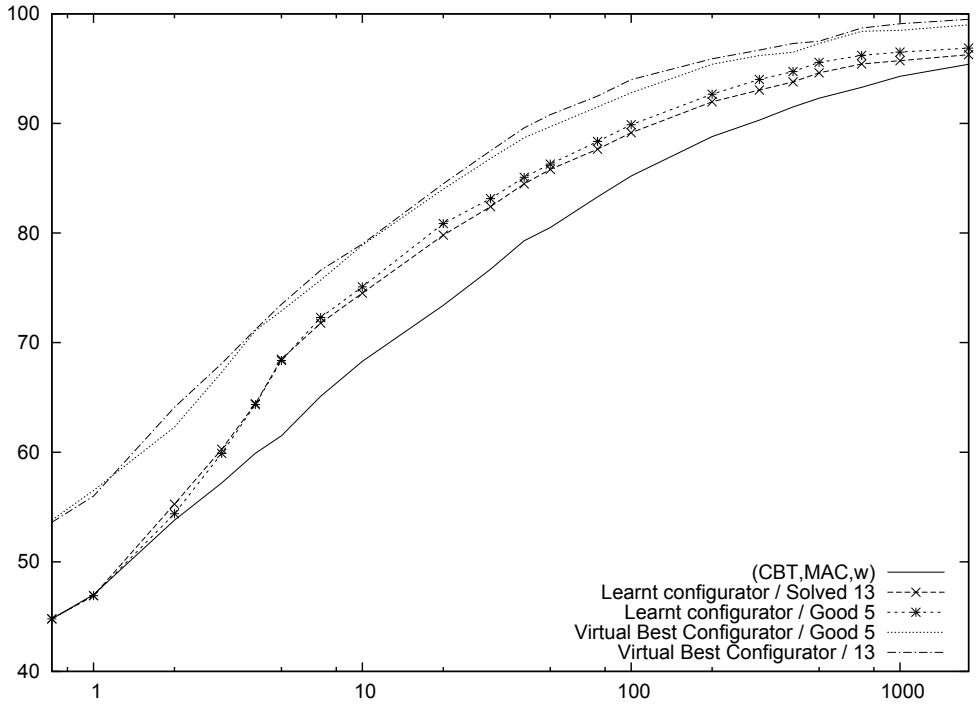


FIGURE 8.1 – *Évolution du pourcentage d'instances résolues en fonction du temps CPU (en secondes)*

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Chapitre 9

Conclusion

Sommaire

9.1 Bilan 97
9.2 Perspectives 98

9.1 Bilan

Nous avons proposé dans cette thèse un solveur générique pour pouvoir comparer des algorithmes issus de l'état de l'art – et de nouvelles propositions – de résolution de problèmes de satisfaction de contraintes. Nous avons collecté plus de mille instances et effectué une étude expérimentale importante en tenant compte du non déterminisme des configurations de notre solveur générique. Ce faisant nous avons souligné que treize configurations seulement sont nécessaires pour couvrir notre benchmark, c'est-à-dire que nous avons toujours une configuration statistiquement aussi bonne que la meilleure configuration pour chaque instance. Nous avons proposé une nouvelle méthode de choix d'un sous-ensemble de configurations pour servir de base à un sélecteur de configuration. Notre proposition est de maximiser le nombre d'instances pour lesquelles il y a une bonne configuration disponible ; ce que nous comparons avec une méthode déjà présentée dans [AGM13] visant à maximiser le nombre d'instances résolues par au moins une configuration. Nous montrons que notre approche apporte de meilleur résultats à la fois pour un meilleur sélecteur virtuel et en pratique ; ces travaux ont été publiés à la conférence *Principles and Practice of Constraint Programming 2014* [BNS14].

Cadre générique paramétrable Nous avons décrit toutes les composantes de notre cadre générique. Il possède quatre paramètres qui désignent le mécanisme de retour arrière, le type de propagation, l'heuristique de choix de variable et l'utilisation ou non d'une décomposition arborescente comme BTD. Nous avons élargi le cadre de [LBH04] en y ajoutant DR, CBJR comme mécanismes de retour arrière et BTD comme paramètre orthogonal.

Analyse expérimentale des différentes configurations Une première évaluation expérimentale nous a permis de constater que les mécanismes de retour-arrière intelli-

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

gents tels que CBJ et DR sont redondants par rapport à une exploitation de décomposition arborescente, de sorte que la combinaison de ces deux mécanismes n'apporte rien. Nous avons donc éliminé les configurations combinant ces mécanismes.

Nous avons lancé les vingt-quatre configurations restantes de notre cadre générique quinze fois sur chacune des 1092 instances de notre benchmark. Nous avons défini ce qu'était une configuration bonne pour une instance à l'aide de tests statistiques. Cela nous a permis de détecter que certaines configurations sont dominées car il y a pour chaque instance une configuration statistiquement meilleure que la dominée. Il y a treize configurations complémentaires non dominées.

Nous avons montré qu'il n'y avait pas de critère évident pour décider si la décomposition était désirable pour une instance donnée.

Méthode de choix de portfolio Nous avons listé les propriétés des instances que nous fournissons à nos sélecteurs de configurations pour les entraîner. Le point mis en avant ici est la sélection du sous-ensemble de configurations parmi lesquelles le sélecteur peut choisir. Nous comparons théoriquement deux méthodes, une visant à maximiser le nombre d'instances résolues par au moins une configuration et une autre visant à maximiser le nombre d'instances pour lesquelles il y a une bonne configuration disponible. Nous choisissons ces ensembles avec un algorithme glouton. La comparaison des meilleurs solveurs virtuels souligne bien les buts de chaque stratégie.

Résultats expérimentaux des différents sélecteurs de configurations Nous avons enfin comparé expérimentalement les différents sélecteurs. Nous comparons à la fois le nombre de fois où une bonne configuration est sélectionnée et également le taux de succès des sélecteurs. La stratégie que nous proposons, visant à proposer de bonnes configurations pour le plus d'instances possibles, obtient de meilleurs résultats (légèrement en ce qui concerne le taux de succès) dans les deux cas.

9.2 Perspectives

Ces travaux pourraient être poursuivis sur plusieurs plans. Chaque composante pourrait être revisitée et approfondie. L'ensemble d'instances peut être enrichi ou changé pour étudier la robustesse de nos propositions. De la même façon, Patrick Prosser m'a conseillé de tester plusieurs implémentations sur plusieurs architectures pour garantir encore une fois la robustesse de ces travaux.

Une autre piste est d'augmenter les possibilités offertes par notre solveur générique que ce soit en y ajoutant des heuristiques de choix de variables comme les Impacts ou Activity, ou tester d'autres heuristiques pour DR. L'extension aux CSP n-aires est aussi désirable.

Nous pourrions comparer nos algorithmes gloutons pour les stratégies de sélection aux résultats optimaux (et donc résoudre sans approximation le problème de couverture d'ensemble), pour voir si notre algorithme change les résultats. Quoiqu'il en soit, les deux stratégies remplissant leurs objectifs respectifs, ce serait une simple curiosité.

Nous avons également d'autres pistes de stratégies de sélection, qui seraient des compromis entre les stratégies *Solved* et *Good* donnant différents poids aux différents sous-objectifs de chaque stratégie. Cela pourrait permettre plus de souplesse dans la sélection des algorithmes.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Bibliographie

[AGM13] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An empirical evaluation of portfolios approaches for solving csps. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 316–324. Springer, 2013.

[AST09] Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *Principles and Practice of Constraint Programming-CP 2009*, pages 142–157. Springer, 2009.

[Bak94] Andrew B. Baker. The hazards of fancy backtracking. In *AAAI*, pages 288–293, 1994.

[BB13] Roberto Battiti and Mauro Brunato. *The LION Way : Machine Learning plus Intelligent Optimization.* Lionsolver inc., 2013.

[BCP+07] Christian Bessiere, Remi Coletta, Thierry Petit, et al. Learning implied global constraints. In *IJCAI*, pages 44–49, 2007.

[BHLS04] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *ECAI*, volume 16, page 146, 2004.

[BNS14] Loïc Blet, Samba Ndojh Ndiaye, and Christine Solnon. Experimental comparison of BTD and intelligent backtracking : Towards an automatic per-instance algorithm selector. In *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 190–206. Springer, 2014.

[BR96] Christian Bessiere and Jean-Charles Régin. Mac and combined heuristics : Two reasons to forsake fc (and cbj ?) on hard problems. In *Principles and Practice of Constraint Programming–CP96*, pages 61–75. Springer, 1996.

[BSPV02] Mauro Birattari, Thomas Stützle, Luis Paquete, and Klaus Varrentrapp. A racing algorithm for configuring metaheuristics. In *GECCO*, volume 2, pages 11–18. Citeseer, 2002.

[CH67] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on*, 13(1) :21–27, 1967.

[CM02] A. Cornuéjols and L. Miclet. *Apprentissage artificiel : concepts et algorithmes.* Algorithmes (Paris). Eyrolles, 2002.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

[CV95] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3) :273–297, 1995.

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

[Dar01] A. Darwiche. Recursive conditioning. *Artificial Intelligence*, 126 :5–41, 2001.

[DB10] Marco Dorigo and Mauro Birattari. Ant colony optimization. In *Encyclopedia of machine learning*, pages 36–39. Springer, 2010.

[DGSV06] Simon De Givry, Thomas Schiex, and Gerard Verfaillie. Exploiting tree decomposition and soft local consistency in weighted csp. In *AAAI*, volume 6, pages 1–6, 2006.

[DM07] R. Dechter and R. Mateescu. AND/OR search spaces for graphical models. *Artificial Intelligence*, 171 :73–106, 2007.

[DP87] R. Dechter and J. Pearl. The Cycle-cutset method for Improving Search Performance in AI Applications. In *Proceedings of the third IEEE on Artificial Intelligence Applications*, pages 224–230, 1987.

[DP89] R. Dechter and J. Pearl. Tree-Clustering for Constraint Networks. *Artificial Intelligence*, 38 :353–366, 1989.

[ESWR96] Hani El Sakkout, Mark G Wallace, and E Barry Richards. An instance of adaptive constraint propagation. In *Principles and Practice of Constraint Programming–CP96*, pages 164–178. Springer, 1996.

[FLP14] Jean-Guillaume Fages, Xavier Lorca, and Thierry Petit. Self-decomposable global constraints. In *ECAI 2014 - 21st European Conference on Artificial Intelligence,18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, pages 297–302, 2014.

[FQ85] E. Freuder and M. Quinn. Taking Advantage of Stable Sets of Variables in Constraint Satisfaction Problems. In *Proceedings of the ninth International Joint Conference on Artificial Intelligence*, pages 1076–1078, 1985.

[Fre78] E. Freuder. Synthesizing constraint expressions. *CACM*, 21(11) :958–966, 1978.

[FWI+98] Eibe Frank, Yong Wang, Stuart Inglis, Geoffrey Holmes, and Ian H Witten. Using model trees for classification. *Machine Learning*, 32(1) :63–76, 1998.

[Gin93] Matthew Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1 :25–46, 1993.

[GL97] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[GLS00] G. Gottlob, N. Leone, and F. Scarcello. A Comparison of Structural CSP Decomposition Methods. *Artificial Intelligence*, 124 :343–282, 2000.

[GS01] Carla P Gomes and Bart Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1) :43–62, 2001.

[Har75] John A. Hartigan. *Clustering Algorithms*. John Wiley & Sons, Inc., New York, NY, USA, 99th edition, 1975.

[HDW94] Geoffrey Holmes, Andrew Donkin, and Ian H Witten. Weka : A machine learning workbench. In *Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on*, pages 357–361. IEEE, 1994.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

[HE04] Greg Hamerly and Charles Elkan. Learning the k in k-means. *Advances in neural information processing systems*, 16 :281, 2004.

[HFH+09] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software : an update. *ACM SIGKDD Explorations Newsletter*, 11(1) :10–18, 2009.

[HGJ+14] Bilal Hussain, Ian P. Gent, Christopher A. Jefferson, Lars Kotthoff, Ian Miguel, Glenna F. Nightingale, and Peter Nightingale. Discriminating instance generation for automated constraint model selection. In *CP*, September 2014.

[HHLBS09] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils : an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36(1) :267–306, 2009.

[HKMO14] Barry Hurley, Lars Kotthoff, Yuri Malitsky, and Barry O'Sullivan. Proteus : A hierarchical portfolio of solvers and transformations. In *CPAIOR*, May 2014.

[Hoo12] Holger H. Hoos. Programming by optimization. *Communications of the ACM*, 55 :70–80, February 2012.

[JL02] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artif. Intell.*, 139(1) :21–45, 2002.

[JNT05] P. Jégou, S. N. Ndiaye, and C. Terrioux. Computing and exploiting tree-decompositions for solving constraint networks. In *Proceedings of CP*, pages 777–781, 2005.

[JNT06] P. Jégou, S. N. Ndiaye, and C. Terrioux. Strategies and Heuristics for Exploiting Tree-decompositions of Constraint Networks. In *Inference methods based on graphical structures of knowledge (WIGSK'06), ECAI workshop*, pages 13–18, 2006.

[JNT07a] P. Jégou, S.N. Ndiaye, and C. Terrioux. Dynamic Management of Heuristics for Solving Structured CSPs. In *Proceedings of 13th International Conference on Principles and Practice of Constraint Programming (CP-2007)*, pages 364–378, 2007.

[JNT07b] Philippe Jégou, Samba Ndojh Ndiaye, and Cyril Terrioux. Dynamic management of heuristics for solving structured csps. In *Proceedings of the 13th international conference on Principles and practice of constraint programming*, CP'07, pages 364–378, Berlin, Heidelberg, 2007. Springer-Verlag.

[JT03a] P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146 :43–75, 2003.

[JT03b] Philippe Jégou and Cyril Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artif. Intell.*, 146 :43–75, May 2003.

[Kja90] Uffe Kjaerulff. Triangulation of graphs - algorithms giving small total state space. Technical report, Judex R.R. Aalborg., Denmark, 1990.

[KMS+11] Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm selection and scheduling. In *Principles and Practice of Constraint Programming–CP 2011*, pages 454–469. Springer, 2011.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

[KMST10] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. Isac-instance-specific algorithm configuration. In *ECAI*, volume 215, pages 751–756, 2010.

[LA87] P. J. M. Laarhoven and E. H. L. Aarts, editors. *Simulated annealing : theory and applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[LBH04] Christophe Lecoutre, Frederic Boussemart, and Fred Hemery. Backjump-based techniques versus conflict-directed heuristics. In *Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on*, pages 549–557. IEEE, 2004.

[Lec09] Christophe Lecoutre. *Constraint Networks : Techniques and Algorithms*. Wiley-IEEE Press, 2009.

[LKLM13] Giovanni Di Liberto, Serdar Kadioglu, Kevin Leo, and Yuri Malitsky. Dash : Dynamic approach for switching heuristics. *CoRR*, abs/1307.4689, 2013.

[MH97] Nenad Mladenovic and Pierre Hansen. Variable neighborhood search. *Computers & OR*, 24(11) :1097–1100, 1997.

[MJT13] Achref El Mouelhi, Philippe Jégou, and Cyril Terrioux. Microstructures for csps with constraints of arbitrary arity. In *Proceedings of the Tenth Symposium on Abstraction, Reformulation, and Approximation, SARA 2013, 11-12 July 2013, Leavenworth, Was hington, USA.*, 2013.

[MMG11] Massimo Morara, Jacopo Mauro, and Maurizio Gabbrielli. Solving XCSP problems by using Gecode. *arXiv preprint arXiv :1112.6096*, 2011.

[MVH11] Laurent D. Michel and Pascal Van Hentenryck. Activity-based search for black-box contraint-programming solvers. Technical report, 2011.

[OHH+08] Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Irish Conference on Artificial Intelligence and Cognitive Science*, 2008.

[PV04] Cédric Pralet and Gérard Verfaillie. Travelling in the world of local searches in the space of partial assignments. In *CPAIOR*, pages 240–255, 2004.

[Qui93] John Ross Quinlan. *C4.5 : programs for machine learning*, volume 1. Morgan Kaufmann, 1993.

[Ref04] Philippe Refalo. Impact-based search strategies for constraint programming. In *Principles and Practice of Constraint Programming–CP 2004*, pages 557–571. Springer, 2004.

[Ric76] John R Rice. The algorithm selection problem. *Advances in Computers*, 15 :65–118, 1976.

[RS86] N. Robertson and P.D. Seymour. Graph minors II : Algorithmic aspects of treewidth. *Algorithms*, 7 :309–322, 1986.

[SD96] Barbara M Smith and Martin E Dyer. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81(1) :155–181, 1996.

[SKC94] B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 337 – 343. MIT Press, 1994.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

[Wal75] D. Waltz. Understanding line drawings of scenes with shadows. In *The Psychology of Computer Vision*, pages 19–91. P. H. Winston, McGraw–Hill, New York, 1975.

[XHHL12] Lin Xu, Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. Evaluating component solver contributions to portfolio-based algorithm selectors. In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, pages 228–241, 2012.

[XHHLB08] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Satzilla : Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res.(JAIR)*, 32 :565–606, 2008.

[XHLB10] Lin Xu, Holger Hoos, and Kevin Leyton-Brown. Hydra : Automatically configuring algorithms for portfolio-based selection. In *AAAI*, volume 10, pages 210–216, 2010.

[ZSZM06] Roie Zivan, Uri Shapen, Moshe Zazone, and Amnon Meisels. Retroactive ordering for dynamic backtracking. In *CP*, pages 766–771, 2006.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2015ISAL0085/these.pdf
© [L. Blet], [2015], INSA de Lyon, tous droits réservés