Méthodes à divergences pour la résolution de problèmes de satisfaction de contraintes et d'optimisation combinatoire

Wafa Karoui

To cite this version:

Wafa Karoui. Méthodes à divergences pour la résolution de problèmes de satisfaction de contraintes et d'optimisation combinatoire. Automatique / Robotique. INSA de Toulouse, 2010. Français. NNT : . tel-00538672

HAL Id: tel-00538672
https://theses.hal.science/tel-00538672

Submitted on 29 Nov 2010

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.

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par *l'Institut National des Sciences Appliquées de Toulouse*
Discipline ou spécialité : *Systèmes Informatiques*

Présentée et soutenue par *Wafa KAROUI*
Le *09/10/2010*

Titre :

Méthodes à divergences pour la résolution de problèmes de satisfaction de contraintes et d'optimisation combinatoire

---

JURY

Mohamed MOALLA	*Professeur*
Emmanuel NERON	*Professeur*
Mohamed HAOUARI	*Professeur*
Pierre LOPEZ	*Chargé de recherche*
Jouhaina CHAOUACHI	*Maître de conférences*
Marie-José HUGUET	*Maître de conférences*

Ecole doctorale : *Ecole Doctorale Systèmes (EDSys)*
Unité de recherche : *LAAS-CNRS de Toulouse*
Directeur(s) de Thèse : *Pierre LOPEZ/Marie-José HUGUET/Mohamed HAOUARI*
Rapporteurs : *Emmanuel NERON/Jouhaina CHAOUACHI*

Remerciements

Je remercie très sincèrement M. Pierre Lopez, M. Mohamed Haouari, et Mme Marie-José Huguet, de m'avoir encadrée. Ils m'ont fait découvrir le monde de la recherche. Merci pour votre confiance et pour les multiples discussions qui ont permis l'aboutissement de cette thèse. Je tiens à remercier vivement M. Pierre Lopez pour m'avoir accueillie au sein de l'équipe MOGISA au LAAS-CNRS et M. Mohamed Haouari pour m'avoir accueillie au sein de l'unité ROI de l'EPT.

J'exprime toute ma gratitude à M. Mohamed Moalla qui m'a fait l'honneur de présider le jury de ma soutenance.

Je tiens à remercier M. Emmanuel Néron et Mme Jouhaina Chaouachi qui ont eu la gentillesse d'accepter de rapporter cette thèse. Merci pour vos nombreux conseils.

Merci également aux membres du département informatique de l'université du Mirail et en particulier Mme Caroline Thierry pour sa confiance et ses encouragements.

Je souhaite remercier tous les membres du groupe MOGISA, permanents, doctorants et stagiaires. Ce fut un réel plaisir de travailler à vos cotés durant ces dernières années. Je n'oublie pas non plus les anciens du groupe qui ont continué leurs chemins ailleurs, merci à Oumar, William et bien d'autres encore. Aussi, je tiens à remercier mes amis en dehors du groupe et en particulier Vincent. Je remercie également les membres de l'unité ROI de l'EPT.

Finalement, je tiens à remercier mes parents sans qui rien de tout cela n'aurait été possible. Merci pour leur patience et pour m'avoir permise de développer ce goût pour la science. Merci également au reste de ma famille et particulièrement à ma sœur Olfa.

Que chacun trouve dans mon travail l'expression de ma haute considération.

A la mémoire
de ma grand-mère
Habiba

TABLE DES MATIÈRES

TABLE DES MATIÈRES . . . . **i**

LISTE DES TABLEAUX . . . . **v**

LISTE DES FIGURES . . . . **vi**

INTRODUCTION **1**

CHAPITRE 1 : PROBLÈMES DE SATISFACTION DE CONTRAINTES 5

1.1 Le formalisme des CSP 6

1.1.1 CSP 7

1.1.2 Arité d'une contrainte 8

1.1.3 Instanciation 8

1.1.4 No-goods 8

1.1.5 Satisfaction de contraintes 8

1.1.6 Solution 8

1.1.7 Espace de recherche 9

1.1.8 Représentation graphique d'un CSP 9

1.2 Paradigme général de traitement des CSP 10

1.3 Techniques de propagation 11

1.3.1 Consistance de sommet 12

1.3.2 Consistance d'arc 13

1.3.3 Autres consistances 19

1.4 Heuristiques d'instanciation 19

1.4.1 Généralités 19

1.4.2 Principe de l'échec au plus tôt 20

1.4.3 Heuristique du domaine minimal 20

1.4.4 Heuristiques du degré 20

1.4.5 Heuristiques du degré dynamique 20

1.4.6 Heuristique du degré pondéré 20
1.4.7 Heuristique de la largeur minimale 21
1.4.8 Heuristique du conflit minimum 21
1.5 Méthodes de recherche arborescente 21
1.5.1 Chronogical Backtracking 22
1.5.2 Méthodes de résolution exploitant la propagation 24
1.5.3 Méthodes de résolution exploitant l'apprentissage 28
1.6 Synthèse de l'existant 29

CHAPITRE 2 : MÉTHODES DE RÉSOLUTION AVEC DIVERGENCES 30

2.1 Introduction 31
2.2 Limited Discrepancy Search (LDS) 33
2.3 Improved Limited Discrepancy Search (ILDS) 37
2.4 Depth-bounded Discrepancy Search (DDS) 38
2.5 Interleaved Depth-First Search (IDFS) 40
2.6 Reverse LDS (RLDS) 45
2.7 Depth-Weighted Discrepancy Search (DWDS) 46
2.8 Discrepancy-Bounded Depth First Search (DBDFS) 47
2.9 Conclusion 49

CHAPITRE 3 : HEURISTIQUES ET DIVERGENCES POUR CSP . . . 50

3.1 Introduction 51
3.2 Mécanismes d'amélioration pour les méthodes de recherche arborescente 51
3.2.1 Pondération des variables 52
3.2.2 Exemple illustratif 52
3.2.3 Pondération des valeurs 53
3.2.4 Exemple illustratif 54
3.3 Mécanismes d'amélioration pour les méthodes de recherche à divergences 55
3.3.1 Restriction des divergences 55
3.3.2 Différents modes de comptage des divergences 57
3.3.3 Positionnement des divergences 59

3.4 Combinaison de mécanismes 60
3.4.1 YIELDS 60
3.4.2 Exemple illustratif 62
3.4.3 Exploitation des no-goods 63
3.4.4 Exemple illustratif 64
3.5 Expérimentations 66
3.5.1 Problèmes de *car sequencing* 66
3.5.2 Problèmes aléatoires 71
3.6 Conclusion 80

CHAPITRE 4 : MÉTHODES À DIVERGENCES POUR OPTIMISATION 81
4.1 Méthodes à base de divergences pour l'optimisation 82
4.1.1 Climbing Discrepancy Search (CDS) 82
4.1.2 Exemple illustratif 83
4.1.3 Climbing Depth-bounded Discrepancy Search (CDDS) 84
4.1.4 Autres adaptations de la méthode CDS 85
4.2 Problèmes d'ordonnancement avec contraintes de délais 86
4.2.1 Définitions 86
4.2.2 Modélisation 87
4.2.3 Méthodes de résolution de la littérature 88
4.3 Divergences pour les time-lags 88
4.3.1 Adaptation de la méthode CDS 88
4.3.2 Positionnement des divergences 89
4.3.3 Heuristique d'initialisation utilisée 89
4.3.4 Pondération des variables 90
4.3.5 Exploitation des pondérations des variables 92
4.4 Expérimentations 94
4.4.1 Benchmarks 94
4.4.2 Résultats obtenus 94
4.5 Conclusion 98

CONCLUSION GÉNÉRALE & PERSPECTIVES **99**

BIBLIOGRAPHIE **102**

LISTE DES TABLEAUX

1.I Tableau récapitulatif des complexités 15

3.I CB - Temps limite 200 s 68

3.II LDS - Comptage binaire - Temps limite 200 s 69

3.III LDS - Comptage non-binaire - Temps limite 200 s 69

3.IV LDS - Comptage binaire 70

3.V LDS - Comptage non-binaire 70

3.VI LDS - Comptage mixte - VarOrder :`Lexico` 71

3.VII LDS - Comptage mixte - VarOrder :`Wvar`$\oplus$`Lexico` 71

4.I Comparaison des heuristiques 95

4.II Comparaison des variantes proposées 95

4.III Résultats obtenus sur quelques instances 97

4.IV Comparaison des variantes à divergences limitées par profondeur 98

LISTE DES FIGURES

1.1 Paradigme de la programmation par contraintes 11
1.2 Compromis réduction-résolution de CSP 12
1.3 Comportement de l'algorithme Chronogical Backtracking (CB) 23
1.4 Comportement d'un algorithme utilisant la propagation de contraintes . 25

2.1 Une divergence dans un arbre binaire 31
2.2 Deux modes de comptage des divergences dans un arbre non-binaire . . 32
2.3 Trace d'exécution de l'algorithme OLDS 35
2.4 Trace d'exécution de l'algorithme ILDS 38
2.5 Trace d'exécution de l'algorithme DDS 40
2.6 Trace d'exécution de l'algorithme IDFS 42
2.7 Trace d'exécution de l'algorithme RLDS 46
2.8 Trace d'exécution de l'algorithme DBDFS 48

3.1 Heuristique de poids sur les variables 53
3.2 Pondération des valeurs 55
3.3 Restriction des divergences 57
3.4 Modes binaire et non-binaire pour le comptage des divergences 58
3.5 Arbre de recherche développé par LDS (29 nœuds) 62
3.6 Arbre de recherche développé par YIELDS (21 nœuds) 63
3.7 Matrice initiale du CSP 65
3.8 Matrice du CSP après exploitation du no-good 65
3.9 Arbre de recherche développé par NG-YIELDS (15 nœuds) 66
3.10 Versions de MAC 73
3.11 Heuristiques de variables avec pondération avec comptage binaire . . . 74
3.12 Heuristiques de variables avec pondération avec comptage non-binaire . 75
3.13 Heuristique dom/Wvar dans différentes méthodes 76
3.14 Heuristique dom/wdeg dans différentes méthodes 77
3.15 Comptage mixte 78

4.1 Arbres de recherche développés par CDS 84
4.2 Différentes possibilités d'incrémentation du poids associé à un job . . . 91
4.3 Modes de comptage des poids sur les jobs 93

Liste des Algorithmes

1 NC-1(X, D, C) 13
2 Révise-Domaine($(x_i, x_j), (X, D, C)$) 16
3 AC-1(X, D, C) 16
4 AC-3(X, D, C) 17
5 AC-4(X, D, C) 18
6 Chronogical-Backtracking(X, D, C, Sol) 22
7 Forward-Checking(X, D, C, Sol) 26
8 Update(X, D, C, (x,v)) 26
9 MAC(X, D, C, Sol) 27
10 OLDS-itération 34
11 OLDS 34
12 LDS-itération 36
13 LDS 36
14 ILDS-itération 37
15 DDS-itération 39
16 DDS 39
17 Pure_IDFS 42
18 Limited_IDFS-itération 43
19 Limited_IDFS 44
20 RLDS-itération 45
21 RLDS 45
22 DWDS-itération 47
23 DWDS 47
24 DRestrict(CurrentMaxDiscr,UsedDiscr) 56
25 YIELDS_Framework(Sol, X, D, C) 61
26 YIELDS_Iteration($Sol, X, D, C, W_{var}, W_{val}, CurrentMaxDiscr$) 61
27 CDS(X, D, C, f, Sol) 83

INTRODUCTION

Contexte du travail et problématique

Pour débuter cette introduction et pour situer le contexte de mon travail, je me permettrai de citer quelques passages de présentation du cours de programmation par contraintes (PPC) de Christine Solnon [Solnon 2003] :

“La notion de contrainte est très naturellement présente dans notre vie quotidienne, qu’il s’agisse d’affecter des stages à des étudiants en respectant leurs souhaits, de ranger des pièces de formes diverses dans une boîte rigide, de planifier le trafic aérien pour que tous les avions puissent décoller et atterrir sans se percuter, ou encore d’établir un menu à la fois équilibré et appétissant. La notion de "Problème de Satisfaction de Contraintes" (ou CSP en abrégé, pour *Constraint Satisfaction Problem*) désigne l’ensemble de ces problèmes, définis par des contraintes, et consistant à chercher une solution les respectant. [...] La résolution de CSP est généralement combinatoire dans le sens où il faut envisager un très grand nombre de combinaisons avant d’en trouver une qui satisfasse toutes les contraintes. [...] Bien souvent, la puissance de calcul des ordinateurs ne suffit pas pour examiner toutes les combinaisons possibles en un temps acceptable, et il est nécessaire d’introduire des "raisonnements" et des "heuristiques" permettant de réduire la combinatoire et de guider la recherche vers les bonnes combinaisons.”

Effectivement, les CSP sont généralement NP-complets et on ne connaît pas de méthodes générales pour les résoudre dans des délais acceptables en pratique, sauf pour des problèmes de petite taille. Dans le pire des cas, les méthodes de recherche arborescente qui traitent les CSP consomment un temps qui croît exponentiellement en fonction du nombre de variables d’entrée du problème. Divers principes de recherche ont été proposés pour exploiter les caractéristiques des CSP dans le parcours d’un arbre de recherche afin d’améliorer l’ordre de visite des feuilles, diminuer le nombre de nœuds générés et résoudre les problèmes le plus rapidement possible. L’un des principes de recherche pro-

posé est la recherche à base de divergences qui a vu le jour dans [Harvey 1995b].

Un problème d'ordonnancement consiste à organiser l'exécution d'un ensemble d'activités soumises à des contraintes de temps et de ressources. C'est un problème qui est, contrairement à ce qu'il paraît, l'un des problèmes les plus difficiles fréquemment rencontrés dans la gestion des systèmes de production. On peut distinguer plusieurs familles de problèmes d'ordonnancement. Parmi les familles les plus connues, nous pouvons citer les problèmes d'atelier où il s'agit d'allouer une machine et une date à chaque opération. L'objectif étant toujours d'améliorer la rentabilité, cela peut se traduire de diverses manières : vérifier si l'ensemble des opérations est réalisable avant une certaine date (contexte de satisfiabilité), optimiser une fonction-objectif (par exemple, minimiser la durée d'un projet, minimiser des retards, maximiser un revenu). Quand l'ordre des ressources à visiter par chaque opération est fixé au préalable et qu'il est identique pour tous les jobs (un job est un ensemble d'opérations à exécution séquentielle), on parle de problème de type flowshop. Quand l'ordre des ressources à visiter par chaque opération est fixé au préalable également mais qu'il varie d'un job à un autre, on parle de problème de type jobshop. Nous pouvons citer également les problèmes de car-sequencing qui consistent à ordonnancer une chaîne de fabrication de voitures qui demandent des options différentes.

Plusieurs filières de recherche se sont intéressées à ce type de problèmes. Les travaux visant l'apport d'une meilleure aide à la décision pour ces problèmes se basent sur des fondements théoriques comme ceux de la programmation linéaire en nombres entiers ou de la programmation par contraintes. Le paradigme de la programmation par contraintes a fait ses preuves et continue ses avancées que ce soit pour résoudre des problèmes de satisfaction ou des problèmes d'optimisation combinatoire. Notre travail s'inscrit dans ce cadre pour présenter de nouvelles variantes de méthodes à base de divergences pour la résolution des problèmes de décision.

Contributions

Dans le cadre de cette thèse, nous nous intéressons principalement à l'étude et l'amélioration des méthodes à base de divergences. Pas uniquement toutefois, puisque certaines techniques peuvent être utiles dans des contextes plus généraux et donc pour des méthodes sans divergences, au départ dans un contexte de satisfaction, et puis dans un contexte d'optimisation. L'étude de ces améliorations ne pouvait se faire sans l'étude et l'adaptation à certains problèmes de la littérature : des problèmes d'ordonnancement de type car-sequencing ainsi que des problèmes aléatoires pour les CSP et des problèmes d'ordonnancement de type flowshop et jobshop avec contraintes de délais pour l'optimisation.

Organisation de la thèse

Ainsi, ce rapport est structuré en quatre chapitres. Les deux premiers portent sur un état de l'art des domaines abordés et les deux derniers présentent les contributions que nous apportons pour améliorer les méthodes de résolution basées sur les divergences pour les CSP et pour les problèmes d'optimisation.

Dans le premier chapitre, nous présentons le formalisme CSP et ses différents concepts. Nous présentons également les principales méthodes de propagation, heuristiques et méthodes de recherche arborescente connues pour la résolution des problèmes dans le cadre de ce formalisme.

Le deuxième chapitre se focalise sur les différentes méthodes arborescentes à base de divergences. L'étude et l'imprégnation des principes de ces méthodes nous servira dans la suite pour baser nos contributions et pouvoir les situer parmi les travaux pré-existants.

Le troisième chapitre est consacré à la proposition de nouveaux mécanismes permettant d'améliorer les méthodes de recherche générales en exploitant les échecs ren-

contrés pendant la résolution. D'autres techniques spécifiques aux méthodes à base de divergences sont ensuite proposées. La deuxième partie de ce chapitre est dédiée aux expérimentations numériques. Nous y faisons une analyse des résultats des tests menés sur des benchmarks de la littérature à travers certains indicateurs. Après une description des conditions de tests, nous présentons, puis commentons les résultats obtenus.

Dans le quatrième chapitre, après une présentation de la résolution des problèmes d'optimisation par les méthodes à divergences et des problèmes d'ordonnancement avec contraintes de délais, nous traitons l'adaptation d'une méthode arborescente basée sur la notion de divergence pour la résolution de ces problèmes. Différentes variantes de cette méthode sont étudiées. Nous évaluons ensuite les performances de ces variantes sur des benchmarks issus de la littérature.

Nous terminons ce manuscrit de thèse par une conclusion générale sur le sujet abordé et les avancées obtenues, ainsi que les perspectives envisagées sur une suite à donner à ce travail de recherche.

CHAPITRE 1

LES PROBLÈMES DE SATISFACTION DE CONTRAINTES

DAns ce chapitre, nous présentons le formalisme des "problèmes de satisfaction des contraintes" (CSP) et ses différents concepts. Nous présentons aussi les principales méthodes de propagation, heuristiques et méthodes de recherche arborescente connues pour la résolution des problèmes dans le cadre de ce formalisme.

1.1 Le formalisme des CSP

"Constraint programming (CP) represents one of the closest approaches computer science has yet made to the Holy Grail of programming : the user states the problem, the computer solves it" (La programmation par contraintes (PPC) représente peut-être l'avancée la plus importante que l'informatique ait connue dans la quête du Saint Graal de la programmation : l'utilisateur définit le problème, l'ordinateur le résout) [Freuder 1997].

Le raisonnement par contraintes est aujourd'hui connu pour sa capacité à résoudre de nombreux problèmes combinatoires (coloration de graphes, emploi du temps, allocation de fréquences, etc.). Pour exploiter ce raisonnement, il suffit de décrire ou modéliser le problème considéré sous la forme d'un "Problème de Satisfaction de Contraintes" (ou CSP, pour *Constraint Satisfaction Problem*). La description du problème s'effectue en termes de variables devant être instanciées tout en respectant un certain nombre de contraintes. Intuitivement, la question principale que l'on se pose est donc celle de l'existence des bonnes valeurs et donc d'une solution satisfaisant les contraintes du problème.

Historiquement, le concept de contrainte était déjà utilisé en 1963 dans des travaux de Sutherland [Sutherland 1963]. Dans les années 70, différents langages expérimentaux autour de la notion de contrainte et de résolution de contraintes ont été proposés. Toujours dans les années 70, le concept de CSP a vu le jour grâce à des chercheurs en intelligence artificielle, avec des langages comme Alice [Lauriere 1978], un langage de modélisation et de résolution pionnier, Prolog-II dans les années 80 un des premiers langages de programmation par contraintes reconnus [Colmerauer 1982], puis CHIP à la fin des années 80 [Dincbas 1988], CLP [Jaffar 1994], etc. On parlait alors déjà d'algorithmes de consistance locale. Plusieurs méthodes de résolution comme la méthode "Backtrack" ont été définies [Montanari 1974, Mackworth 1977, Freuder 1978]. Depuis la fin des années 80, la programmation par contraintes (PPC) profitant des avancées faites dans les domaines de l'intelligence artificielle, la recherche opérationnelle, l'algèbre de Boole et la logique propositionnelle a désormais fait ses preuves. Parmi les

systèmes de contraintes commerciaux ou de domaine public, on peut citer ILOG-Solver [Puget 1995], CHIP [Aggoun 1993], ECLIPSE, et CLAIRE [Caseau 1996], ainsi que ECLAIR [Laburthe 1998]. Plusieurs domaines d'application ont été explorés avec succès et plusieurs challenges ont été relevés [Prosser 1993, Esquirol 1995, Apt 2000].

Dans ce paragraphe, nous rappelons quelques concepts fondamentaux des CSP [Tsang 1993, Dechter 2003, Beek 2006].

1.1.1 CSP

Un CSP est un problème modélisé sous la forme d'un ensemble de contraintes posées sur des variables, chacune de ces variables prenant ses valeurs dans un domaine. De façon plus formelle, on définira un CSP par un triplet (X,D,C) tel que :

- $X = \{x_1, x_2, \ldots, x_n\}$ est l'ensemble des variables.
- $D = \{D_1, D_2, \ldots, D_n\}$ est l'ensemble des domaines des valeurs. Chaque domaine de valeurs D_i est un ensemble discret et fini contenant les valeurs possibles pour la variable x_i.
- $C = \{C_1, C_2, \ldots, C_m\}$ est l'ensemble des contraintes du problème. Chaque contrainte C_i porte sur un sous-ensemble des variables $\{x_1, x_2, \ldots, x_n\}$ de X appelé $Var(C_i)$. On appelle $Rel(C_{ij})$ les paires de valeurs de $Var(C_i)$ et $Var(C_j)$ qui sont compatibles.

  Rappelons que les contraintes peuvent être définies aussi bien en extension qu'en intension.

  Contrainte en extension : la contrainte est définie par une liste de tuples. Dans notre cas, la sémantique est telle que $Rel(C_{ij})$ est défini par la liste des paires qui vérifient une certaine relation.

  Contrainte en intension : la contrainte est décrite par un prédicat. Dans notre cas, la sémantique est telle que le prédicat est évalué sur tous les n-uplets candidats. Les n-uplets pour lesquels la relation est évaluée à *true* sont retenus pour former $Rel(C_{ij})$.

1.1.2 Arité d'une contrainte

On désigne par $Var(C_i)$ l'ensemble des variables mises en jeu par la contrainte C_i. Si $Var(C_i)$ est un singleton, on dit que C_i est une contrainte unaire. Si $Var(C_i)$ est une paire alors C_i est une contrainte binaire. Si $Var(C_i)$ est un ensemble à k éléments alors C_i est une contrainte k-aire. On dit aussi que k est l'arité de C_i. On désignera par C_{ij} la contrainte binaire portant sur les variables x_i et x_j. Un CSP est binaire si toutes ses contraintes sont au plus d'arité 2. Les CSP binaires jouent un rôle particulier car n'importe quel CSP peut être transformé en un CSP binaire équivalent.

1.1.3 Instanciation

Une instanciation élémentaire est un couple variable-valeur (x_i, v) tel que v appartient à D_i. Une instanciation est un ensemble d'instanciations élémentaires. Si une instanciation porte sur toutes les variables du CSP alors on parle d'instanciation complète ; dans le cas contraire, on parle d'instanciation partielle.

1.1.4 No-goods

Un "no-good" du CSP P est une instanciation partielle de P qui ne fait partie d'aucune de ses solutions.

Un no-good minimal de P est un no-good qui ne contient aucun n-uplet qui est un no-good de P.

1.1.5 Satisfaction de contraintes

Soit I une instanciation partielle d'un CSP P et soit C_i une contrainte de P. On dit que C_i est satisfaite par I si toutes les variables $Var(C_i)$ sont instanciées par I et si la combinaison des valeurs affectées aux variables de $Var(C_i)$ appartient à $Rel(C_i)$.

1.1.6 Solution

Une solution d'un CSP P est une instanciation complète S telle que toutes les contraintes de P sont satisfaites par S. Si P admet au moins une solution alors P est dit consistant

ou satisfiable ; sinon, P est inconsistant, insatisfiable ou sur-contraint.

1.1.7 Espace de recherche

L'ensemble de toutes les combinaisons de $D_1 \times D_2 \times \ldots \times D_n$ est appelé espace de solutions ou espace de recherche dans la mesure où la solution doit être cherchée dans cet espace. L'espace de recherche d'un CSP est souvent représenté par une arborescence. De ce fait, on appelle les méthodes qui le parcourent des méthodes de recherche arborescente.

1.1.8 Représentation graphique d'un CSP

La structure d'un CSP peut être représentée par divers graphes. Nous présentons ci-dessous quelques-uns d'entre eux.

Graphe de contraintes

Le graphe de contraintes d'un CSP binaire P est un graphe simple (ce n'est pas un multigraphe) où les sommets représentent les variables de P et les arêtes représentent les contraintes de P. Il y a une arête entre les sommets représentant les variables x_i et x_j si et seulement s'il existe une contrainte $C_{ij} \in C$ tel que $Var(C_{ij}) = \{x_i, x_j\}$.

Le concept de graphe de contraintes peut être défini pour des CSP quelconques (non binaires). Les contraintes ne sont alors pas représentées par des arêtes mais par des hyper-arêtes qui relient des sous-ensembles de variables impliquées dans une même contrainte. Dans ce cas, on a un hyper-graphe de contraintes.

On appelle **voisinage** d'un sommet du graphe l'ensemble des sommets adjacents de ce graphe. C'est-à-dire la liste des sommets que l'on peut accéder directement depuis le sommet courant (concept d'adjacence).

Le **degré d'un sommet** du graphe est le nombre d'arêtes qui entrent et qui sortent du sommet en cours. Pour un graphe non orienté comme le graphe de contraintes, ceci

correspond tout naturellement au cardinal de son voisinage. On définit également le **degré d'un graphe** comme étant le maximum des degrés du graphe. Un graphe de degré 4 aura des sommets dont le degré maximum peut être 4.

Etant donné un ordre sur les nœuds d'un graphe, la **largeur d'un nœud** dans un graphe est le nombre des nœuds adjacents et d'ordre inféreieur à ce nœud. Etant donné un ordre sur les nœuds, la **largeur d'un graphe** est la plus grande largeur de ses nœuds.

Graphe de consistance d'un CSP binaire

Contrairement au graphe de contraintes qui ne prend pas en compte la définition des contraintes du CSP, le graphe de consistance représente les combinaisons de valeurs compatibles. Le graphe de consistance d'un CSP binaire P est un graphe simple dont les sommets représentent l'ensemble des valeurs de l'union des domaines $\bigcup_{(i=1,...,n)} D_i$. Deux sommets $u_i \in D_i$, $v_j \in D_j$ sont reliés par une arête si la contrainte $C_{ij} \in C$ et si $(u_i, v_j) \in Rel(c_{ij})$ c'est-à-dire que u_i et v_j sont deux valeurs compatibles.

Graphe d'inconsistance d'un CSP binaire

Le graphe d'inconsistance d'un CSP binaire P est un graphe simple dont les sommets représentent l'ensemble des valeurs $\bigcup_{(i=1,...,n)} D_i$. Deux sommets $u_i \in D_i$, et $v_j \in D_j$ sont reliés par une arête si la contrainte $C_{ij} \in C$ et si $(u_i, v_j) \notin Rel(C_{ij})$ c'est-à-dire si u_i et v_j sont deux valeurs incompatibles.

1.2 Paradigme général de traitement des CSP

Dans le cadre du formalisme CSP, la résolution est assurée par des méthodes qui se basent *a minima* sur :

- des techniques de propagation de contraintes (ou de filtrage) : elles sont associées à des processus de déduction logique permettant d'expliciter des informations sur un CSP ;
- des heuristiques d'instanciation (ordre de choix des variables et des valeurs) ;

– des stratégies de parcours de l'espace de recherche ou arborescence.

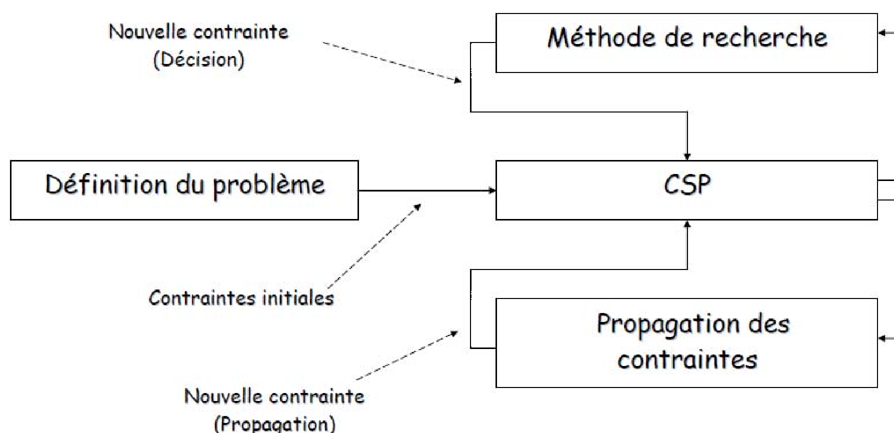


Figure 1.1 – Paradigme de la programmation par contraintes [Baptiste 2000]

A partir d'un CSP initial, une méthode de recherche s'emploie à parcourir l'espace de recherche tout en s'appuyant sur des propagations de contraintes pour élaguer l'espace et réduire le problème initial (ce mécanisme constitue le paradigme de la programmation par contraintes illustré figure 1.1). Le parcours de l'espace de recherche est guidé par des heuristiques de choix de variables et de valeurs à instancier en priorité et par une stratégie d'exploration (en profondeur ou en largeur, avec gestion des échecs, retour arrière chronologique ou non, etc.).

Dans les paragraphes suivants, nous dressons un état de l'art des différentes composantes d'une méthode de résolution de CSP.

1.3 Techniques de propagation

Dans ce paragraphe, nous allons mettre en évidence l'intérêt de la propagation de contraintes et parcourir les techniques de réduction de CSP les plus importantes.

Dans la définition des CSP, il est fréquent que les domaines contiennent des valeurs qui ne font partie d'aucune solution. L'élimination de telles valeurs n'a aucun effet sur l'ensemble des solutions, mais est pourtant utile du point de vue de l'efficacité de la résolution. Il en est de même pour les contraintes, où certains n-uplets ne font partie d'aucune

solution. En réduisant un domaine de valeur et/ou une relation définissant une contrainte, on obtient un nouveau CSP équivalent au CSP de départ mais de taille réduite. Un CSP est minimal s'il ne peut pas être réduit. La tâche qui consiste à calculer l'équivalent minimal d'un CSP est souvent très coûteuse. Ainsi, en général, il n'est pas envisageable de réduire un CSP à son équivalent minimal avant de le résoudre. Cependant, un certain niveau de réduction peut être très avantageux (figure 1.2).

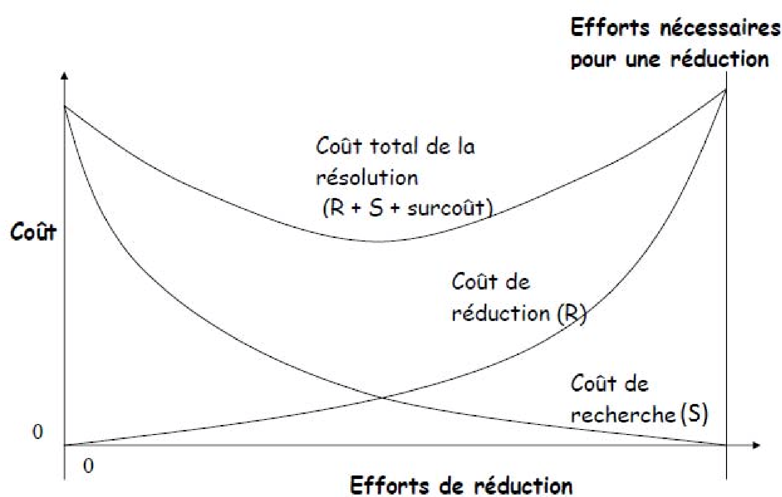


Figure 1.2 – Compromis réduction-résolution de CSP [Tsang 1993]

La réduction d'un CSP peut être faite une seule fois, par exemple en prétraitement avant d'utiliser un algorithme de résolution, ou plusieurs fois pendant l'exploration de l'espace de recherche pour élaguer progressivement l'arbre de recherche.

On utilise par la suite le concept de graphe de contraintes défini précédemment pour introduire les notions de consistance de sommet et de consistance d'arc pour les CSP binaires.

1.3.1 Consistance de sommet

Un CSP est sommet-cohérent (Node-Consistent ou NC), si toutes les contraintes unaires sont satisfaites pour tous les éléments des domaines de valeurs [Mackworth 1977].

L'algorithme de consistance de sommet suivant (algorithme 1) enlève les éléments superflus en vérifiant les domaines de valeurs, l'un après l'autre. Cet algorithme a une complexité de $O(nd)$, où n est le nombre de variables et d la taille maximale des domaines de valeurs. Un CSP est sommet-cohérent si et seulement si tous les sommets de son graphe de contraintes sont sommet-cohérents.

Algorithme 1 : NC-1(X, D, C)

```
1 pour x_i ∈ X faire
2     pour v_i ∈ D_i faire
3         si ∃c_i ∈ C alors
4             si (var(c_i) = {x_i}) et (v_i ∉ Rel(c_i)) alors
5                 D_i ← D_i \ {v_i}
```

1.3.2 Consistance d'arc

Un arc (x_i, x_j) du graphe de contraintes d'un CSP binaire $P = (X, D, C)$ est dit arc-cohérent (Arc-Consistent ou AC) si et seulement si, pour toute valeur $v_i \in D_i$ qui satisfait toutes les contraintes unaires sur x_i, il existe $v_j \in D_j$ telle que $(v_i, v_j) \in Rel(C_{ij})$. Un CSP est arc-cohérent si et seulement si tous les arcs de son graphe de contraintes sont arc-cohérents [Mackworth 1977].

Algorithmes de consistance d'arc

La consistance d'arc [Mackworth 1977] est la consistance locale la plus largement utilisée. Plusieurs algorithmes de filtrage par consistance d'arc ont été proposés pour les CSP binaires. En 1977, Mackworth propose les algorithmes AC-1, AC-2 et surtout AC-3 [Mackworth 1985] qui est le meilleur des trois (*i.e.* complexité en temps en $O(ed^3)$, et en espace en $O(ed^2)$ dans le pire des cas)[1]. Depuis, de nombreuses améliorations ont suivi avec comme objectif la réduction des complexités spatiale et/ou temporelle. La première AC-4 [Mohr 1986], (*i.e.,* a une complexité en temps en $O(ed^2)$, et en es-

[1]On note e le nombre de contraintes, n le nombre de variables et d la taille du plus grand domaine.

pace en $O(ed^2)$ dans le pire des cas), introduit la notion de support mais avec un comportement en moyenne moins bon que AC-3. Deux algorithmes AC-5 ont été proposés [Deville 1991, Hentenryck 1992a, Perlin 1992] : ces algorithmes améliorent la complexité spatiale de AC-4 (*i.e.,* en $O(ed)$) pour des types de contraintes particulières mais se réduisent à AC-3 ou AC-4 dans le cas général. La proposition de AC-6 [Bessière 1994] (*i.e.,* avec des complexités optimales en temps en $O(ed^2)$ et en espace en $O(ed)$) améliore en moyenne le comportement de AC-3 mais conserve la complexité dans le pire des cas de AC-4. AC-6++, une amélioration de AC-6 est proposée par Bessière et Régin. [Bessière 1995] introduit AC-Inference, un algorithme qui utilise des métaconnaissances pour réduire le nombre de tests et [Bessière 1999] propose AC-7, un algorithme de consistance d'arc général exploitant la même idée. Une bonne implémentation de cet algorithme permet d'atteindre des complexités linéaires intéressantes (*i.e.,* complexité spatiale dans le pire des cas en $O(ed)$). En se basant sur l'exploitation des supports minimaux, comme pour AC-6 mais sans les mémoriser, [Chmeiss 1998] propose AC-8. Cet algorithme reste d'un intérêt limité contrairement à son analogue pour la consistance de chemin PC-8. En 2001, Bessière *et al.* propose AC-2000 et AC-2001 basées sur AC-3, alliant simplicité et efficacité [Bessière 2001]. Ces deux algorithmes ciblent toutefois des problèmes relativement peu contraints, leur efficacité étant mise en défaut pour des propagations en trop grand nombre. La même année Zhang propose AC-3.1, autre amélioration de AC-3, simple, efficace, et d'application plus large [Zhang 2001]. En 2005, une amélioration de AC-3 et AC-2001 est proposée par [Mehta 2005]. Cette dernière se base sur la révision des supports des valeurs et réduit les temps de résolution de 50%. En 2005, Bessière *et al.* proposent AC-2001/3.1 [Bessière 2005] qui a la particularité d'avoir une complexité temporelle dans le pire des cas optimale, tout en restant simple. Les expérimentations montrent que cet algorithme est compétitif avec AC-6. D'autres améliorations basées sur l'étude des résidus sont proposées par [Lecoutre 2007a, Lecoutre 2008].

L'extension de la consistance d'arc au CSP n-aires est communément appelée consistance d'arc généralisée (GAC) [Mohr 1987, Mohr 1988]. Quelques travaux ont été réalisés sur des algorithmes de filtrage par consistance d'arc généralisée. Mackworth propose l'algorithme CN qui est une généralisation de AC-3 aux contraintes non-binaires

Algorithme	Complexité en temps	Complexité en espace
NC-1 [Mackworth 1977]	$O(nd)$	$O(nd)$
AC-1 [Mackworth 1977]	au pire : $O(ned^3)$	$O(e+nd)$
AC-3 [Mackworth 1977]	borne inf. : $\Omega(ed^2)$ borne sup. : $O(ed^3)$	$O(e+nd)$
AC-4 [Mohr 1986]	au pire : $O(ed^2)$	$O(ed^2)$
AC-5 [Hentenryck 1992a]	$O(ed^2)$	$O(ed)$
AC-6 [Bessière 1993]	$O(ed^2)$	$O(ed)$
AC-7 [Bessière 1995]	$O(ed^2)$	$O(ed)$
AC-2000 [Bessière 2001]	$O(ed^3)$	$O(ed)$
AC-2001 [Bessière 2001]	$O(ed^2)$	$O(ed)$
CN(GAC-3) [Mackworth 1977]	$O(ea^2d^{a+1})$	$O(ea+nd)$
GAC-4 [Mohr 1988]	$O(ed^a)$	$O(ed^a+nd)$
GAC-schema [Bessière 1997]	$O(ed^a)$	$O(eda^2)$

Tableau 1.I – Tableau récapitulatif des complexités en temps et en espace de quelques algorithmes de NC et d'AC pour CSP binaires et n-aires. e = le nombre de contraintes du CSP ; n = le nombre de variables du CSP ; d = la taille du plus grand domaine du CSP ; a = la plus grande arité des contraintes du CSP.

[Mackworth 1977]. [Mohr 1987, Mohr 1988] proposent l'algorithme GAC-4 qui est une généralisation de AC-4 aux contraintes non-binaires. Vu leurs mauvaises complexités, ces deux algorithmes sont très rarement utilisés en pratique. [Bessière 1997] propose GAC-schema, une généralisation de AC-7 qui permet de traiter des contraintes d'arité quelconques exprimés en extension, sous forme d'expressions arithmétiques ou encore sous forme de prédicats sans sémantique particulière. Cet algorithme est plus performant que GAC-4 du fait qu'il ne mémorise que les supports utiles (*cf.* table 1.I pour une récapitulation des complexités).

Dans la suite, nous présentons le principe de quelques-uns des plus basiques de ces algorithmes : AC-1, AC-3 et AC-4.

AC-1 et AC-3

Le filtrage par arc-consistance consiste à retirer de chaque domaine D_i toutes les valeurs qui ne satisfont pas les contraintes binaires C_{ij} compte tenu des domaines D_j.

Ceci est réalisé par la fonction Révise-Domaine (algorithme 2).

Algorithme 2 : Révise-Domaine$((x_i, x_j), (X, D, C))$

---

1 éliminé ← faux
2 **pour** $v_i \in D_i$ **faire**
3 **si** $\nexists\, v_j \in D_j$ *et* $(v_i, v_j) \in$ *Rel*(C_{ij}) **alors**
4     $D_i \leftarrow D_i \setminus \{ v_i \}$
5   éliminé ← vrai
6 **retourner** éliminé

Le premier algorithme proposé, AC-1 (algorithme 3) pour la réalisation de la consistance d'arc fait tout d'abord appel à la procédure NC-1 pour réaliser la consistance de sommet, puis il fait appel à la fonction Révise-Domaine qui réalise la consistance arc par arc. Avec AC-1, tout retrait d'une valeur du domaine d'une variable entraîne le réexamen complet de toutes les contraintes. La complexité temporelle de AC-1 est en $O(ned^3)$ et sa complexité spatiale est en $O(e+nd)$ où e est le nombre de contraintes binaires, n le nombre de variables et d la taille maximale des domaines de valeurs.

Algorithme 3 : AC-1(X, D, C)

---

1 NC-1(X, D, C)
2 E ← $\{(x_i, x_j) \,/\, C_{ij} \in C\}$ *% E est l'ensemble de tous les arcs du graphe de contraintes*
3 modifié ← vrai
4 **tant que** *modifié* **faire**
5   modifié ← faux
6   **pour** $(x_i, x_j) \in E$ **faire**
7   modifié ← modifié ou Révise-Domaine((x_i,x_j), (X, D, C))

L'algorithme AC-3 (algorithme 4) améliore AC-1 : il ne réexamine pas toutes les contraintes à chaque suppression de valeur. Une contrainte ne sera de nouveau examinée que si elle peut être affectée par la révision du domaine d'une variable. La complexité en temps d'AC-3 est en $O(ed^3)$, sa complexité en espace mémoire est en $O(e+nd)$.

Algorithme 4 : AC-3(X, D, C)

```
1 NC-1(X, D, C)
2 E ← {(xi,xj) / Cij ∈ C}
3 tant que E ≠ ∅ faire
4 |  E ← E \ (xi,xj)
5 |  si Révise-Domaine((xi,xj), (X, D, C)) alors
6 |  |  E ← E ⋃{(xk,xi) / Cki ∈ C et k ≠ j}
```

AC-4

Des améliorations de l'algorithme AC-3 sont apportées dans AC-4 qui est un algorithme optimal en temps pour la réalisation de la consistance d'arc (algorithme 5). L'idée sur laquelle s'appuie AC-4 utilise la notion de support. On dit qu'une valeur (x_i, u) est un support pour (x_j, v) si $C_{ij} \in C$ et si (x_i, u) est compatible avec (x_j, v). Une valeur (x_i, u) est viable (non éliminable) si elle admet un support dans chacun des domaines de valeurs des variables qui sont en contrainte avec x_i. Quand une valeur (x_i, u) est éliminée, il n'est pas nécessaire d'examiner la répercussion de cette élimination sur toutes les autres valeurs du problème. En effet, on peut se limiter à la vérification de la viabilité des valeurs qui sont supportées par (x_i, u).

Pour identifier les valeurs dont la viabilité devrait être re-examinée, AC-4 utilise trois informations supplémentaires :

- pour chaque valeur (x_i, u), AC-4 maintient l'ensemble de toutes les valeurs supportées par (x_i, u). Cet ensemble est noté S dans l'algorithme de AC-4.
- un tableau de compteurs (Ctr), qui sert à compter le nombre de supports par valeur et par contrainte. Ainsi $Ctr[(i, j), u]$ mémorise le nombre de supports de (x_i, u) dans le domaine de valeurs de x_j, (ce compteur n'existera que si $C_{ij} \in C$). Quand un compteur est réduit à 0, la valeur correspondante est supprimée.
- une matrice booléenne M qui sert à marquer les valeurs éliminées. Si une valeur (x_i, u) est éliminée, alors $M[i, u]$ sera mis à 1.

La première étape de AC-4 initialise la matrice M, l'ensemble des supports S, et le tableau de compteurs Ctr. Cette étape est composée d'une boucle sur l'ensemble des

Algorithme 5 : AC-4(X, D, C)

```
1  M ← 0 % M matrice booléenne de marquage des valeurs éliminées
2  S ← ∅ % S l'ensemble des supports
3  pour chaque C_ij ∈ C faire
4      pour chaque b ∈ D_i faire
5          Total ← 0
6          pour chaque c ∈ D_j faire
7              si (b,c) ∈ Rel(C_ij) alors
8                  Total ← Total+1
9                  S(x_j,c) ← S(x_j,c) ⋃{(x_i,b)}
10         si Total = 0 alors
11             M[i,b]← 1
12             D_i ← D_i\{(x_i,b)}
13         Ctr[(i,j),b]← Total
14 % fin des initialisations
15 List ← {(x_i,b) / M[i,b]=1}
16 tant que List ≠ ∅ faire
17     List ← List \{(x_j,c)}
18     pour chaque (x_i,b) ∈ S(x_j,c) faire
19         Ctr[(i,j),b] ← Ctr[(i,j),b]-1
20         si (Ctr[(i,j),b]=0) et (M[i,b]=0) alors
21             M[i,b]← 1
22             D_i ← D_i\{(x_i,b)}
23             List ← List ⋃{(x_i,b)}
```

contraintes et de deux boucles sur les domaines de valeurs. Sa complexité est en $O(ed^2)$. La deuxième étape réalise la consistance d'arc en éliminant les valeurs qui n'ont pas de support, c'est-à-dire celles dont le compteur est réduit à 0. La complexité temporelle de la deuxième étape de AC-4 peut être mesurée en comptant le nombre de fois où les compteurs sont réduits. Les compteurs prennent des valeurs positives ayant un maximum de d. Donc, chaque compteur peut être réduit d fois. Puisque un CSP contient $e \times d$ compteurs, le nombre de réductions maximal est de ed^2. La complexité temporelle de AC-4 est donc en $O(ed^2)$.

En plus de l'espace mémoire nécessaire pour le stockage des données du problème,

AC-4 a besoin de mémoriser S, M et Ctr. C'est l'ensemble des supports (S) qui demande le plus d'espace mémoire. Si e_i est le nombre de variables reliées à x_i par une contrainte alors il y a, au plus, ed^2 éléments dans S. D'où une complexité en $O(ed^2)$.

1.3.3 Autres consistances

D'autres types de consistances existent comme la consistance de chemin [Montanari 1974], la consistance d'ordre supérieur ou k-consistance [Freuder 1978] qui est une généralisation de la consistance d'arc et de la consistance de chemin, la consistance de chemin restreinte [Berlandier 1995], les consistances singletons [Debruyne 1997, Bessière 2010], les consistences inverses [Freuder 1996], les consistances relationnelles [Beek 1995] et les algorithmes de filtrage spécifiques généralement proposés pour les contraintes dites globales [Régin 1994, Beldiceanu 1994] [Bessière 2006].

Nous ne détaillerons pas davantage ces méthodes.

1.4 Heuristiques d'instanciation

1.4.1 Généralités

Les heuristiques d'instanciation sont des règles qui déterminent l'ordre suivant lequel on va choisir les variables pour les instancier ainsi que l'ordre suivant lequel on va choisir les valeurs pour les variables. Ainsi, une heuristique d'instanciation est composée :

- d'un ordre sur les variables ;
- d'un ordre sur les valeurs.

Une bonne heuristique d'instanciation peut avoir un grand impact sur la résolution d'un problème de décision et permettre d'accélérer l'obtention de solutions (la détection d'échecs en cas de problèmes insolubles) pendant l'exploration de l'espace de recherche. Ainsi, on évite que de grandes parties de l'arbre de recherche soient explorées en vain. De plus, si l'on veut favoriser l'apparition d'une solution, les variables et les valeurs sont ordonnées par l'heuristique de telle façon que, si une solution existe, elle sera associée à un chemin qui se trouve, par convention, le plus à gauche possible dans l'arbre de

recherche et sera ainsi atteinte au plus tôt lors d'une exploration en profondeur d'abord. Plusieurs travaux se sont interessés aux heuristiques, que ce soit dans un contexte SAT, CSP ou optimisation combinatoire [Hooker 2000, Lecoutre 2007b].

Une heuristique peut être statique ou dynamique selon que les ordres de variables et de valeurs sont fixés au préalable ou évoluent pendant la résolution, de telle sorte qu'elle soit affectée par les changements que subit l'espace de recherche.

Nous présentons ci-dessous quelques exemples d'heuristiques.

1.4.2 Principe de l'échec au plus tôt ou fail-first

Le principe de l'échec au plus tôt ou *fail-first* de Haralick se résume en cette citation : "*To suceed, try first where you are most likely to fail*" (Pour réussir, commence par essayer où il est plus probable d'échouer) [Haralick 1980].

1.4.3 Heuristique du domaine minimal (min-domain)

Cette heuristique dynamique d'ordre sur les variables sélectionne en premier la variable qui a le plus petit domaine de valeurs.

1.4.4 Heuristiques du degré (degree ou deg)

L'heuristique basée sur les degrés [Dechter 1989] ordonne les variables de manière décroissante en fonction de leur degré (*cf.* paragraphe 1.1.8).

1.4.5 Heuristiques du degré dynamique (dynamic-degree ou ddeg)

L'heuristique du degré dynamique ordonne les variables de manière décroissante en fonction de leur degrés dynamiques qui évoluent au cours de la recherche.

1.4.6 Heuristique du degré pondéré (weighted-degree ou wdeg)

A chaque étape de résolution, l'objectif de cette heuristique de variables dynamique [Boussemart 2004a, Boussemart 2004b] est de s'appuyer sur des informations issues des étapes précédentes du processus de recherche de solution. Pour cela, un poids est associé

à chaque contrainte et à chaque fois que le domaine d'une variable devient vide, le poids de la contrainte ayant entrainé cet échec est incrémenté. L'heuristique wdeg choisit alors la variable associée aux contraintes de plus fort poids.

1.4.7 Heuristique de la largeur minimale (minimal-width ou width)

L'heuristique de la largeur minimale [Freuder 1982] ordonne les variables de manière à limiter la largeur du graphe de contraintes (*cf.* paragraphe 1.1.8).

1.4.8 Heuristique du conflit minimum (min-conflict)

Contrairement aux heuristiques présentées ci-dessus, l'heuristique du conflit minimum ordonne les valeurs. Il s'agit de choisir une valeur qui permettra à l'instanciation partielle courante d'être étendue à une solution. Une mesure commune pour juger vraisemblable la participation d'une valeur à une solution est de considérer le nombre de valeurs qui ne sont pas en conflit avec la valeur en question. Ainsi, la valeur qui est compatible avec la plupart des valeurs des variables non instanciées est essayée en premier.

Ces différentes heuristiques sont également combinées entre elles, on retrouve classiquement l'utilisation de l'heuristique d'ordre sur les variable min-domain $\oplus$ ddeg ou min-domain/ddeg ou encore min-domain/wdeg. Les algorithmes de résolution qui utilisent des heuristiques d'ordre assurent, dans la plupart des cas, une résolution plus rapide que celle obtenue par des algorithmes sans heuristique d'ordre. Même si les différentes heuristiques d'ordre sont en général incomparables entre elles, dans de nombreux cas (et notamment sur des problèmes structurés) l'heuristique min-domain/wdeg fournit de très bons résultats.

1.5 Méthodes de recherche arborescente

Les algorithmes qui seront exposés dans ce paragraphe sont des méthodes de résolution dites complètes : si un CSP admet une solution alors ces algorithmes finissent

toujours par la trouver. Ils procèdent à une exploration systématique de l'espace de recherche (en général, suivant une stratégie de parcours en profondeur d'abord) en vue de construire une instanciation complète qui soit cohérente avec toutes les contraintes du CSP.

1.5.1 Chronogical Backtracking

La méthode de retour arrière chronologique (Chronogical Backtracking ou CB) considère les variables l'une après l'autre. Pour chaque variable, elle cherche une valeur qui soit compatible avec toutes les valeurs données aux variables instanciées précédemment. Si une telle valeur existe, alors elle passe à la variable suivante. Sinon, elle revient sur la dernière variable instanciée pour essayer une autre valeur, et ainsi de suite, jusqu'à ce qu'elle trouve une solution ou que tout l'espace de recherche soit exploré (algorithme 6 et figure 1.3).

Une alternative à ce retour arrière chronologique est l'utilisation de divergences. Cette voie sera abordée en détail dans le chapitre suivant.

Algorithme 6 : Chronogical-Backtracking(X, D, C, Sol)

```
1  si InstanciationComplète(Sol) alors
2  |  retourner Sol
3  sinon
4  |  Choisir x ∈ X % sur la base d'une heuristique d'ordre sur les variables
5  |  pour chaque v ∈ D_x faire
6  |  |  si Sol ⋃{(x,v)} satisfait C alors
7  |  |  Sol ← Chronogical-Backtrack(X\{x}, D, C, Sol ⋃{(x,v)})
8  |  |  si InstanciationComplète(Sol) alors
9  |  |  |  retourner Sol
10 |  retourner ∅
```

L'algorithme CB peut être amélioré en reconsidérant plusieurs points, comme la propagation de contraintes [Debruyne 1998] afin de réduire l'arbre de recherche. Il peut être amélioré également en intégrant différentes formes d'apprentissage [Cambazard 2006] ou encore en forçant des "redémarrages" ou "restarts", technique répandue pour les mé-

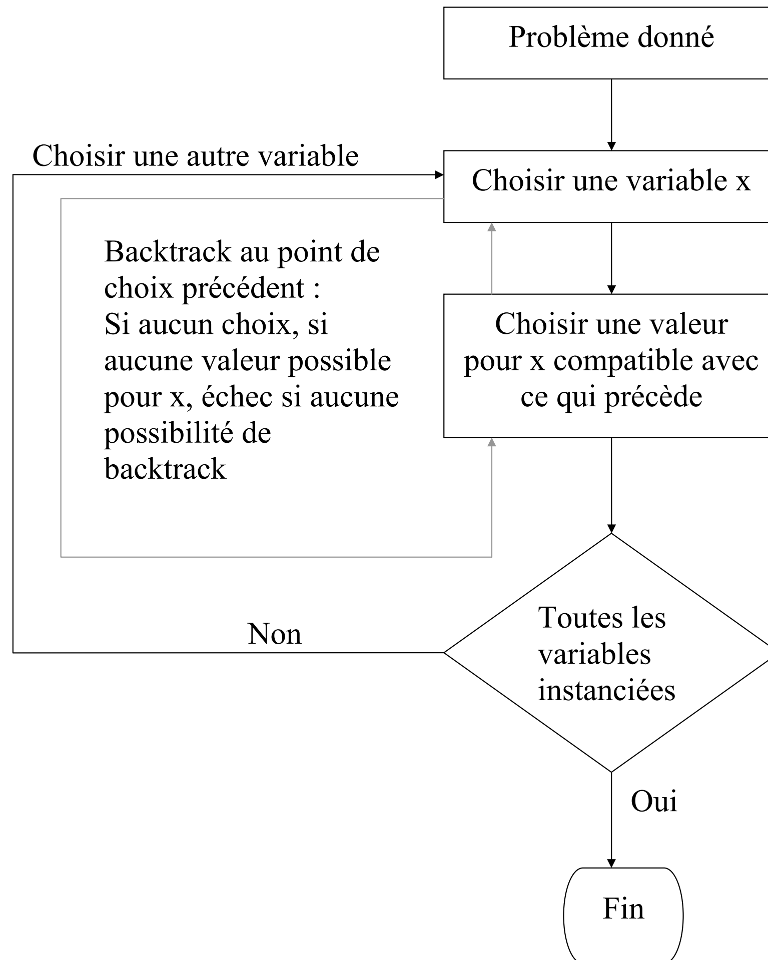


Figure 1.3 – Comportement de l'algorithme Chronogical Backtracking (CB)

thodes de recherche locale. La technique de restart permet de diversifier le parcours de l'espace de recherche en y introduisant de l'aléatoire et conduit en pratique à de bonnes performances.

Dans les paragraphes suivants, nous détaillons quelques méthodes qui répondent à quelques-uns des schémas d'amélioration décrits ci-dessus.

1.5.2 Méthodes de résolution exploitant la propagation

Supposons qu'en cherchant une solution, l'algorithme CB donne à une variable x une valeur v qui exclut toutes les valeurs possibles pour une variable y. L'algorithme CB ne se rendra compte de ce fait que lorsque la variable y est considérée. De plus, avec un retour arrière chronologique, avant que x ne soit reconsidérée, il se peut qu'une grande partie de l'espace de recherche soit explorée en vain. Ceci peut être évité en prenant en compte le fait que (x, v) ne peut faire partie d'une solution, puisqu'il n'y a aucune valeur de y qui soit compatible avec (x, v).

Les algorithmes qui utilisent la propagation de contraintes évitent la situation décrite ci-dessus, en n'acceptant une valeur v pour une variable x qu'après avoir vérifié les domaines des variables futures reliées à x. S'il existe un domaine qui ne contient que des valeurs incompatibles avec v, alors v est éliminée. Par ailleurs, une réduction du problème peut avoir lieu après chaque instanciation (x, v). En effet, les valeurs des variables futures qui ne sont pas compatibles avec (x, v) peuvent être éliminées sans perte de solutions (figure 1.4).

Dans les paragraphes qui suivent, nous allons présenter *Forward-Checking* et *Maintaining Arc-Consistency* qui sont les deux méthodes de résolution de CSP utilisant la propagation de contraintes les plus connus.

Forward-Checking

Chaque fois que la méthode Forward-Checking (FC) instancie une variable, elle élimine des domaines de valeurs des variables liées par une contrainte à la variable instanciée, toutes les valeurs qui ne sont pas compatibles (d'après les contraintes du CSP)

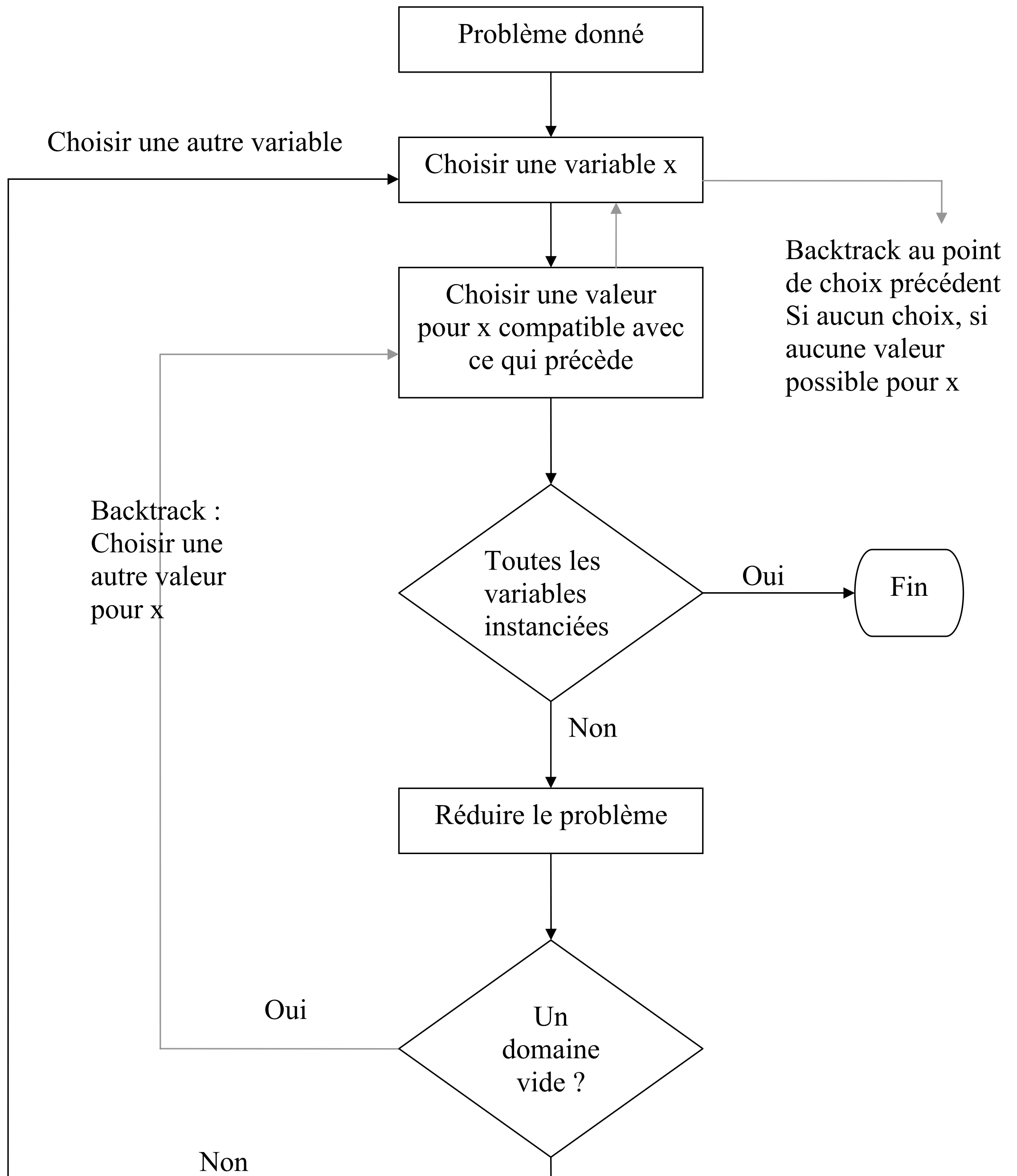


Figure 1.4 – Comportement d’un algorithme utilisant la propagation de contraintes

avec l'instanciation qui vient d'être effectuée. Dans la fonction *Update* (algorithme 8), les domaines des variables voisines de la variable x selon le graphe de contraintes (*cf.* paragraphe 1.1.8) sont réduits de telle sorte que seules les valeurs compatibles avec la valeur de x sont gardées (algorithme 7).

Algorithme 7 : Forward-Checking(X, D, C, Sol)

```
1  si InstanciationComplète(Sol) alors
2  |  retourner Sol
3  sinon
4  Choisir x ∈ X
5  pour chaque v ∈ D_x faire
6  |  si Sol ⋃{(x,v)} satisfait C alors
7  |  |  D' ← Update(X\{x}, D, C, (x,v))
8  |  |  si aucun domaine de D' n'est vide alors
9  |  |  |  Sol ← Forward-Checking(X\{x}, D, C, Sol ⋃{(x,v)})
10 |  |  |  si InstanciationComplète(Sol) alors
11 |  |  |  |  retourner Sol
12 retourner ∅
```

Algorithme 8 : Update(X, D, C, (x,v))

```
1  pour y ∈ X faire
2  |  pour u ∈ D_y faire
3  |  |  si (y,u) est incompatible avec (x,v) alors
4  |  |  |  D_y ← D_y\{u}
5  retourner D
```

Maintaining Arc-Consistency

En consacrant plus de temps de calcul à la réduction de problème, il est possible d'éliminer davantage de valeurs pendant la résolution. En effet avec l'algorithme FC, seules sont éliminées les valeurs des variables voisines de la variable instanciée incompatibles avec l'instanciation courante. Or, il est possible d'éliminer plus de valeurs en

maintenant la consistance d'arc à chaque nœud de l'espace de recherche. Ceci est réalisé par l'algorithme MAC (*Maintening Arc-Consistency* [Sabin 1994]).

Algorithme 9 : MAC(X, D, C, Sol)

```
1  si InstanciationComplète(Sol) alors
2  |  retourner Sol
3  sinon
4  |  choisir x ∈ X
5  |  D'_x ← D_x
6  |  tant que D'_x ≠ ∅ faire
7  |  |  D'_x ← D'_x\{v}
8  |  |  si Sol ⋃{(x,v)} satisfait C alors
9  |  |  |  D" ← Update(X\{x}, D, C, (x,v))
10 |  |  |  si aucun domaine de D" n'est vide alors
11 |  |  |  |  AC(X\x, D", C)
12 |  |  |  |  si aucun domaine de D" n'est vide alors
13 |  |  |  |  |  Sol ← MAC(X\{x}, D", C, Sol ⋃{(x,v)})
14 |  |  |  |  |  si InstanciationComplète(Sol) alors
15 |  |  |  |  |  |  retourner Sol
16 |  |  AC(X, D\D_x ⋃D'_x, C)
17 |  |  si un domaine de D\D_x ⋃D'_x est vide alors
18 |  |  |  retourner ∅
19 |  retourner ∅
```

Cet algorithme (algorithme 9) établit de la consistance d'arc après chaque instanciation de variable. La consistance d'arc est réalisée par l'un des algorithmes de consistance d'arc vus dans le paragraphe 1.3.2 (AC-3, AC-4, AC-6, etc.). On obtient ainsi plusieurs versions de l'algorithme MAC qui diffèrent uniquement par l'algorithme de consistance d'arc utilisé. D'ailleurs, les algorithmes de type AC sont, généralement, évalués dans MAC. Améliorer MAC a donc longtemps été synonyme d'amélioration de AC. En 2004, Likitvivatanavong *et al.* [Likitvivatanavong 2004] proposent d'étudier des stratégies plus performantes de MAC, indépendamment de toute amélioration de AC. Dans ce cadre, ils proposent une amélioration de MAC3 (*i.e.,* MAC basé sur AC-3) s'appuyant sur l'idée d'un cache pour sauvegarder les informations utiles aux propagations de telle sorte d'évi-

ter la redondance des calculs et conservant ainsi toute la simplicité de AC-3. Ils proposent également deux autres versions, *MAC3.1residue* et *MAC3.1resOpt*, basés sur une notion de “résidu du support”. En raison de structures de données supplémentaires de ces deux dernières versions, celles-ci semblent s’effacer au bénéfice de *MAC3cache*.

La fonction *Update*, appelée par MAC, est la même que celle utilisée par l’algorithme Forward-Checking. L’appel de l’algorithme de consistance d’arc dans la procédure Maintaining Arc Consistency a pour effet de rendre le CSP initial arc-cohérent. Le premier appel à l’algorithme de consistance d’arc dans la fonction MAC sert à rétablir la consistance d’arc après d’éventuelles éliminations de valeurs effectuées dans la fonction *Update*. Tandis que le deuxième appel rétablit la consistance d’arc après l’élimination de la valeur v si cette dernière ne donne pas de solution après l’avoir essayée.

1.5.3 Méthodes de résolution exploitant l’apprentissage

Dans ce paragraphe, nous présentons quelques méthodes qui vont collecter des informations au cours des différentes étapes de résolution pour essayer de les exploiter afin d’améliorer les heuristiques d’instanciation en particulier et de guider la recherche en général, que ce soit dans la même itération ou dans les itérations suivantes dans le cas de méthodes itératives ou avec restarts. On peut considérer que ces méthodes mettent en œuvre une sorte d’apprentissage.

Last Conflict reasoning (LC)

La méthode LC [Lecoutre 2006] se base sur un raisonnement sur les conflits pour guider le développement de l’arborescence de recherche vers les variables à l’origine de l’inconsistance. A chaque étape de résolution, l’heuristique de choix de variable sélectionne celle(s) ayant précédemment entrainé des échecs. Son efficacité a été attestée dans le cadre d’une méthode MAC sur des problèmes aléatoires ainsi que sur des problèmes réels. Cette méthode a été étendue par Grimes et Wallace [Grimes 2006] en introduisant des restarts à la méthode de base.

Impact-Based-Search (IBS)

La méthode IBS [Refalo 2004] est une méthode de recherche basée sur une technique issue du domaine de la programmation linéaire en nombres entiers. Dans IBS, l'impact de l'instanciation d'une variable, c'est-à-dire de la réduction de l'espace de recherche qu'elle engendre, est mesuré et utilisé pour privilégier certaines variables lors des prochaines instanciations. La notion d'impact est inspirée du pseudo-coût de la programmation en nombres entiers. Ce dernier mesure l'importance de l'affectation associée. L'efficacité de cette méthode a été attestée sur des problèmes de sac-à-dos multidimensionnel, des carrés magiques et des carrés latins. L'ajustement correct des paramètres ainsi qu'un couplage avec des restarts peut mener à de très bons résultats.

1.6 Synthèse de l'existant

Ce chapitre présente le formalisme CSP avec ses concepts de base et parcourt différentes parties d'une méthode de résolution : les techniques de propagation, les heuristiques de choix d'ordre sur les variables et les valeurs, et les méthodes de parcours de l'espace de recherche. Il est à noter qu'un mécanisme considéré à part, que ce soit la propagation, l'heuristique ou la méthode de parcours, ne peut être évalué dans l'absolu. On ne peut l'étudier que dans l'ensemble, dans son interaction avec les différents autres mécanismes.

Ce chapitre présente aussi quelques méthodes de résolution de base ainsi que quelques améliorations qui exploitent le concept de propagation, et pour d'autres celui de l'apprentissage.

Le catalogue des méthodes présentées dans ce chapitre n'a aucune prétention d'exhaustivité. On pourrait notamment citer encore bien d'autres méthodes de recherche arborescente utilisant différents mécanismes. Par exemple, comme nous l'avons déjà dit, nous examinerons dans le chapitre suivant un type de méthode exploitant une stratégie de parcours de l'arbre de recherche basée sur le principe des divergences.

CHAPITRE 2

MÉTHODES DE RÉSOLUTION DES PROBLÈMES DE SATISFACTION DE CONTRAINTES BASÉES SUR LES DIVERGENCES

DAns ce chapitre, nous présentons différentes méthodes de résolution des CSP basées sur les divergences. Nous donnons le principe de chaque méthode considérée que nous illustrons sur des exemples simples.

2.1 Introduction

Les méthodes à base de divergences définissent une stratégie alternative de parcours de l'arborescence de recherche par rapport aux méthodes basées sur une stratégie de parcours de type Backtrack Chronologique vues au chapitre précédent.

Pour introduire la notion de divergence, il faut au préalable indiquer que cette notion n'a de sens qu'en rapport à la définition d'une heuristique d'instanciation de référence. Une heuristique d'instanciation détermine une politique d'affectation de couples (variable, valeur) qui s'appuie sur la définition d'ordres sur les variables et d'ordres sur les valeurs. Une divergence s'exprime alors comme la contradiction d'un choix privilégié par l'heuristique d'instanciation (*cf.* figure 2.1). En effet, une telle heuristique d'instanciation, que nous appellerons "heuristique" dans la suite pour simplifier, est par définition susceptible de commettre des erreurs ; il se révèle être parfois avantageux de ne pas la suivre.

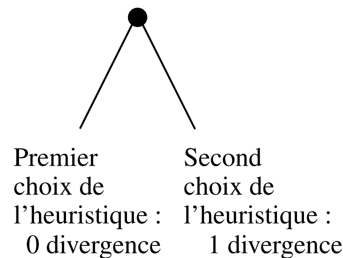


Figure 2.1 – Une divergence dans un arbre binaire

La notion de divergence a été définie initialement pour les arbres à variables binaires appelés également arbres binaires. Dans le cas d'un arbre binaire, on adopte la convention d'associer pour chaque nœud, le fils gauche au premier choix de l'heuristique, le fils droit représentant donc une divergence. Dans le cas d'un arbre n-aire, pour chaque nœud le fils le plus à gauche représente le premier choix de l'heuristique, tous les autres fils correspondent à des divergences. Deux modes de comptage des divergences peuvent

ainsi être envisagés :

- soit le premier fils (à gauche) correspond à 0 divergence et tous les autres fils à 1 divergence. On se ramène ainsi à un comptage binaire des divergences (*cf.* partie gauche de la figure 2.2) ;
- soit le premier fils (à gauche) correspond à 0 divergence et les autres fils sont associés à un nombre de divergence croissant au fur et à mesure qu'on se déplace vers la droite dans l'arbre (*cf.* partie droite de la figure 2.2). On procède alors à un comptage non binaire des divergences pour chaque nœud.

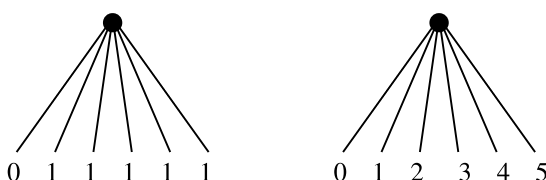


Figure 2.2 – Deux modes de comptage des divergences dans un arbre non-binaire

Initialement, la notion de divergence a été proposée en effectuant des divergences en haut de l'arbre de recherche d'abord. On peut également privilégier les divergences en bas d'abord. Selon le choix de la position des divergences à privilégier et à tester en premier, l'impact est nettement observable sur le parcours de l'espace de recherche.

Depuis la proposition du concept de divergence et de la première méthode de recherche arborescente basée sur ce principe, *Limited Discrepancy Search* (LDS) [Harvey 1995a] [Harvey 1995b], plusieurs travaux se sont appuyés sur ce concept pour développer des méthodes qui exploitent et cherchent à améliorer la méthode de base. Ces méthodes ont été conçues dans le but d'éviter les erreurs faites dans l'arbre de recherche et de donner de nouvelles orientations au parcours de l'espace de recherche en autorisant un certain nombre de divergences. Le nombre de divergences permises représente le nombre de fois où l'on s'écarte du choix privilégié par l'heuristique pour la recherche d'une solution admissible.

Ces méthodes ont été proposées pour répondre à différents objectifs de résolution

(satisfaction ou optimisation) en explorant divers types d'arbres de recherche (binaire et non-binaire). Dans ce chapitre, nous présentons les principales de ces méthodes pour la résolution de CSP.

Notons que pour l'utilisation des méthodes à base de divergence avec propagation, nous adoptons le schéma de comptage qui suit :
Si une ou plusieurs valeurs sont éliminées par propagation suite aux précédentes instanciations, nous ne comptons plus ces valeurs et le comptage des divergences fonctionne comme si ces valeurs n'existaient pas. Bien entendu, quand on remet en question l'instanciation qui les a éliminées, ces valeurs entrent de nouveau dans les calculs des divergences. Nous justifions notre choix par le fait que la divergence est définie comme étant un écart du choix de l'heuristique et que les valeurs éliminées n'auraient pas pu, dans ce contexte, faire partie des choix de l'heuristique.

2.2 Limited Discrepancy Search (LDS)

Le concept de divergence a été proposé initialement au sein de la méthode LDS [Harvey 1995a] [Harvey 1995b]. Dans la méthode LDS, lorsque l'heuristique ne peut aboutir à une solution, on s'autorise lors d'une prochaine étape à s'écarter des choix préconisés pour certaines variables. Une hypothèse faite pour le développement de la méthode LDS est que l'heuristique d'instanciation est suffisamment performante pour guider la recherche de solution dans le parcours de l'arbre en suggérant de bonnes valeurs à sélectionner en premier. S'il y a des erreurs commises par cette heuristique, elles sont en petit nombre et effectuer progressivement des divergences par rapport à cette heuristique va permettre d'aboutir rapidement à une solution admissible.

Les algorithmes 10 et 11 détaillent le principe de la méthode initiale appelée ici *Original LDS* (OLDS) s'appliquant sur un arbre binaire (*i.e.,* à variables binaires). Dans cet algorithme, *est une feuille* teste si un nœud donné est une feuille et donc solution. *Successeurs* retourne les fils du nœud passé en paramètre. *fils-gauche* et *fils-droit* retournent respectivement le premier et le second fils d'un nœud sachant qu'on suppose que l'heuristique considérée classe les fils de gauche à droite. *Profondeur-max* étant

Algorithme 10 : OLDS-itération(nœud, k, Sol, X, D, C)

```
1  si InstanciationComplète(nœud) alors
2  |  retourner nœud
3  s ← Successeurs (nœud)
4  si s = ∅ alors
5  |  retourner ∅
6  si k = 0 alors
7  |  retourner OLDS-itération(fils-gauche(s),0, Sol, X, D, C)
8  sinon
9  |  Sol ← OLDS-itération(fils-droit(s),k-1, Sol, X, D, C)
10 |  si InstanciationComplète(Sol) alors
11 |  |  retourner Sol
12 |  retourner OLDS-itération(fils-gauche(s),k)
```

Algorithme 11 : OLDS(X, D, C)

```
1  pour k ← 0 jusqu'à Profondeur-max faire
2  |  Sol ← OLDS-itération(nœud,k,Sol, X, D, C)
3  |  si InstanciationComplète(Sol) alors
4  |  |  retourner Sol
5  retourner ∅
```

le nombre de variables, et donc la profondeur maximale, les itérations sont au nombre de *Profondeur-max*+1 car, l'arbre étant binaire, on peut diverger une fois au niveau de chaque variable. La fonction *OLDS-itération* considère initialement le nœud racine de l'arbre de recherche associé à un nombre de divergences autorisé k et visite ses successeurs. Tant qu'il n'y a pas de solution, le nombre de divergences est incrémenté, ce qui permet des itérations correspondant à des arbres de recherche de plus en plus grands jusqu'à ce que tout l'espace de recherche soit entièrement visité.

Comme on peut le voir sur la trace d'exécution décrite par la figure 2.3, OLDS diverge prioritairement dans les niveaux les plus hauts de l'arbre en parcourant l'arbre de droite à gauche dans une même itération et en visitant les feuilles de l'arbre suivant un ordre croissant de divergences d'une itération à une autre. Dans cette figure, le nombre

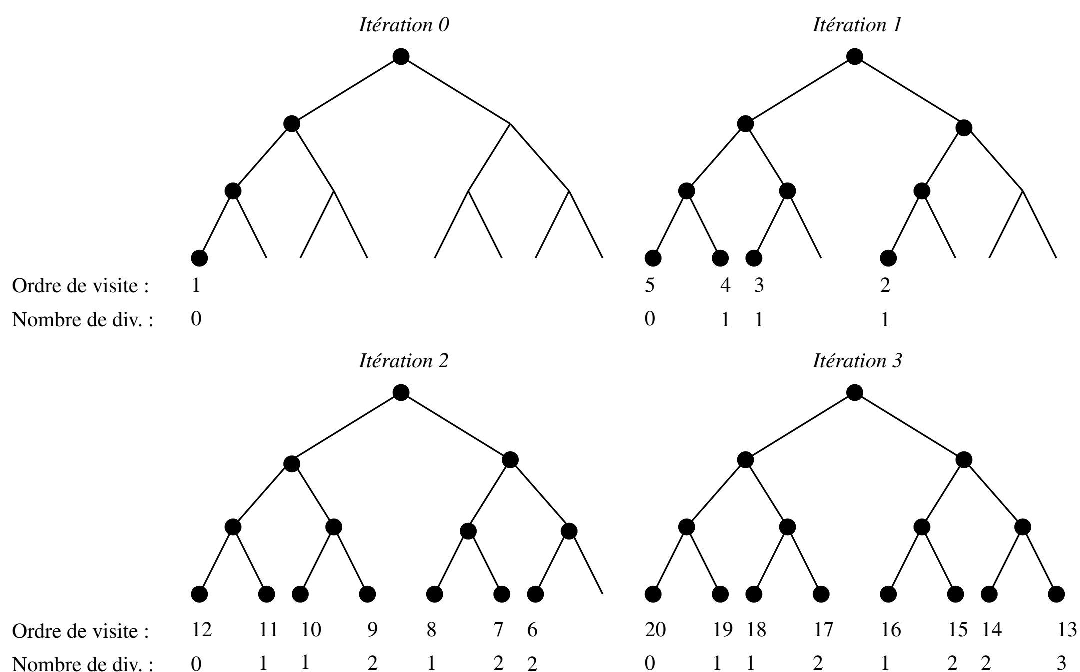


Figure 2.3 – Trace d'exécution de l'algorithme OLDS

indiqué en dessous de chaque feuille, sur la première ligne, correspond à l'ordre de visite des feuilles de l'arbre dans le parcours de l'espace de recherche. Sur la seconde ligne, le nombre correspond au nombre de divergences qui ont permis d'atteindre cette feuille.

La méthode OLDS stoppe dès qu'une solution est trouvée. Lorsque le problème est insoluble, elle explore la totalité de l'arbre de recherche.

Pour la résolution de CSP, on doit bien sûr envisager d'intégrer à OLDS des propagations du type de celles présentes dans Forward-Checking ou AC (*cf.* chapitre précédent). Par la suite, et pour la différencier d'OLDS, nous appellerons LDS la version de la recherche à divergences limitées qui intègre la propagation. Notons que cette nuance n'a pas d'impact sur le mode de comptage des divergences puisque celui-ci est appliqué sur l'arbre de recherche alors que les propagations se font à la suite d'instanciations et affectent uniquement les domaines de valeurs.

Algorithme 12 : LDS-itération(k, Sol, X, D, C)

```
1  si InstanciationComplète(Sol) alors
2  |   retourner Sol
3  sinon
4  |   x ← Heuristique-d'ordre-sur-var(X)
5  |   v ← Heuristique-d'ordre-sur-val(D_x, k)
6  |   si v ≠ premier fils gauche alors
7  |   |   k ← k-1 %on fait une divergence
8  |   si Sol ⋃ {(x,v)} satisfait C alors
9  |   |   D'← Propagation(X\x, D, C, (x,v))
10 |   |   si aucun domaine de D' n'est vide alors
11 |   |   |   Sol ← LDS-itération(k, Sol ⋃ {(x,v)}, X\ {x}, D', C)
12 |   sinon
13 |   |   retourner ∅
```

Algorithme 13 : LDS(X, D, C)

```
1  k ← 0
2  tant que (Sol = ∅) et (k ⪯ k-max) faire
3  |   Sol ← LDS-itération(k, Sol, X, D, C)
4  |   k ← k+1
5  retourner Sol
```

Les algorithmes 12 et 13 récapitulent le principe de LDS. Dans ces algorithmes, *k-max* désigne le nombre maximal de divergences possible dans l'arbre de recherche considéré, qui est fonction de la profondeur de l'arbre (nombre de variables) ainsi que de la nature des variables (binaires ou non-binaires) et du choix adopté pour compter les divergences. *Heuristique-d'ordre-sur-var* et *Heuristique-d'ordre-sur-val* désignent respectivement les heuristiques d'ordre sur les variables et les valeurs. *Propagation* retourne l'ensemble des domaines après avoir propagé les modifications sur le CSP passé en paramètre.

Comme on peut le noter (cf. figure 2.3), la méthode LDS présente beaucoup de redondances. En effet, à chaque itération, LDS visite non seulement les branches qui vérifient le nombre de divergences fixé mais également toutes les branches qui ont un nombre de divergences inférieur. Pour remédier à ces inconvénients, plusieurs adaptations ont été

proposées. Nous les présentons dans la suite.

2.3 Improved Limited Discrepancy Search (ILDS)

On peut remarquer que, dans un arbre binaire, si l'on fait abstraction des effets des éventuelles propagations, il est facile de prévoir le nombre de divergences que l'on peut encore réaliser si l'on considère la profondeur du nœud traité. La méthode ILDS [Korf 1996] ou *recherche à divergence limitée améliorée* exploite cette remarque pour améliorer LDS dans le sens où chaque itération ne traite que les branches qui ont exactement le nombre de divergences ciblé.

Cette méthode est intéressante pour un arbre binaire. Si l'arbre considéré est binaire et a une hauteur d uniforme pour toutes ses branches, alors la complexité asymtotique de l'algorithme de recherche appliqué à cet arbre est réduite de $O(((d+2)/2)*2^d)$ pour LDS à $O(2^d)$ pour ILDS [Korf 1996]. Ce gain n'est pas aussi intéressant si les branches de l'arbre n'atteignent pas toutes cette hauteur et que la hauteur n'est pas uniforme pour toutes les branches, ou bien si l'arbre n'est pas binaire. L'amélioration se base sur la comparaison de la profondeur éventuelle que pourra atteindre la branche en cours et le nombre de divergences autorisé. Si l'on n'a aucune chance de faire toutes ces divergences et que la profondeur restante est plus petite que le nombre de divergences fixé alors il faut abandonner. Cette amélioration s'avère beaucoup moins efficace dans une généralisation à des arbres n-aires pour lesquels l'information sur la hauteur de l'arbre devient clairement moins importante. Elle peut aussi être fonction du choix de comptage des divergences adopté.

Algorithme 14 : ILDS-itération(nœud, k, profondeur, Sol, X, D, C)

```
1 si InstanciationComplète(nœud) alors
2     retourner nœud
3 si profondeur > k alors
4     Sol ← ILDS-itération(fils-gauche(s),k,profondeur-1, Sol, X, D, C)
5 si k > 0 alors
6     Sol ← ILDS-itération(fils-froit(s),k-1,profondeur-1, Sol, X, D, C)
```

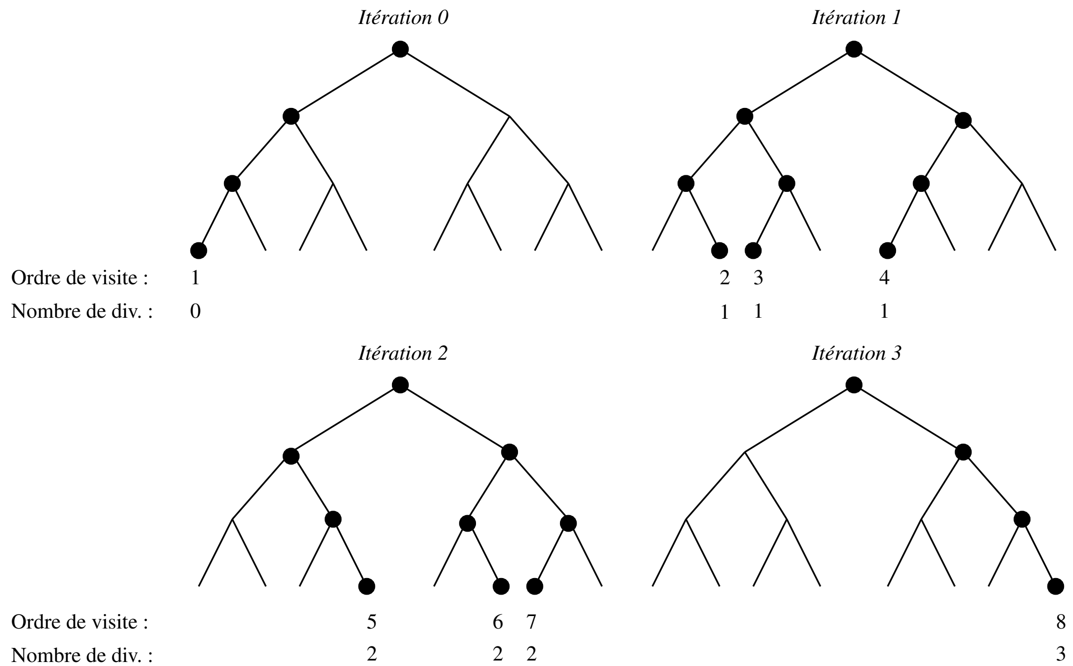


Figure 2.4 – Trace d'exécution de l'algorithme ILDS

L'algorithme de base de ILDS étant identique à ceux de OLDS et LDS dans sa gestion des divergences à part le fait qu'il passe en paramètre la profondeur de l'arbre de recherche, l'algorithme 14 présente la particularité d'une itération de la méthode ILDS. A la différence de OLDS, on effectue un test sur la profondeur du nœud avant toute itération supplémentaire. Comme on peut le voir sur la figure 2.4, contrairement à LDS, la méthode ILDS applique en premier les divergences en bas de l'arborescence. De plus, au sein d'une itération, ILDS parcourt les feuilles de l'arbre de gauche à droite.

2.4 Depth-bounded Discrepancy Search (DDS)

La méthode DDS [Walsh 1997] ou *recherche à divergence limitée en profondeur* est une amélioration de la méthode ILDS. DDS est basée sur le postulat qu'une divergence qui apparaît en bas de l'arbre de recherche est généralement d'un impact moindre qu'une divergence qui apparaît plus haut dans l'arbre. Ainsi, la méthode DDS augmente progressivement le nombre de divergences autorisées en restreignant leurs emplacements dans une partie supérieure de l'arbre paramétrée par une limite sur la profondeur autori-

Algorithme 15 : DDS-itération(nœud, k, Sol, X, D, C)

```
1  si InstanciationComplète(nœud) alors
2  |  retourner (nœud, 0)
3  si k = 0 alors
4  |  (Sol,profondeur) ← DDS-itération(fils-gauche(nœud),0, Sol, X, D, C)
5  |  retourner (Sol, 1+profondeur)
6  si k = 1 alors
7  |  (Sol,profondeur) ← DDS-itération(fils-droit(nœud),0, Sol, X, D, C)
8  |  retourner (Sol, 1+profondeur)
9  si k > 1 alors
10 |  (Sol1,profondeur1) ← DDS-itération(fils-gauche(nœud),k-1, Sol, X, D, C)
11 |  si nœud1 est une feuille alors
12 |  |  retourner (Sol1, 1+profondeur1)
13 |  sinon
14 |  |  (Sol2,profondeur2) ← DDS-itération(fils-droit(nœud),k-1, Sol, X, D, C)
15 |  |  retourner (Sol2, 1+max(profondeur1,profondeur2))
```

Algorithme 16 : DDS(X, D, C)

```
1  k ← 0
2  nœud ← nœud-racine
3  répéter
4  |  (Sol,profondeur) ← DDS-itération(nœud, k, Sol, X, D, C)
5  |  k = k+1
6  jusqu'à InstanciationComplète(Sol) ou k > profondeur;
7  retourner Sol
```

sée pour les divergences. Cette limite sur la profondeur est relâchée au fur et à mesure que la recherche avance et que la contrainte sur le nombre de divergences autorisées est relâchée.

Les algorithmes 15 et 16 présentent le principe de DDS. La figure 2.5 présente la trace de l'exécution de cet algorithme sur un arbre binaire à trois variables. Dans cette figure, la partie supérieure où l'on peut diverger est délimitée par un trait en pointillé. Sur la figure, on note également que les feuilles explorées lors d'une itération donnée de DDS ne sont pas revisitées lors des prochaines itérations comme dans le cadre de la méthode ILDS. On remarque également que lors d'une itération de DDS, des solutions

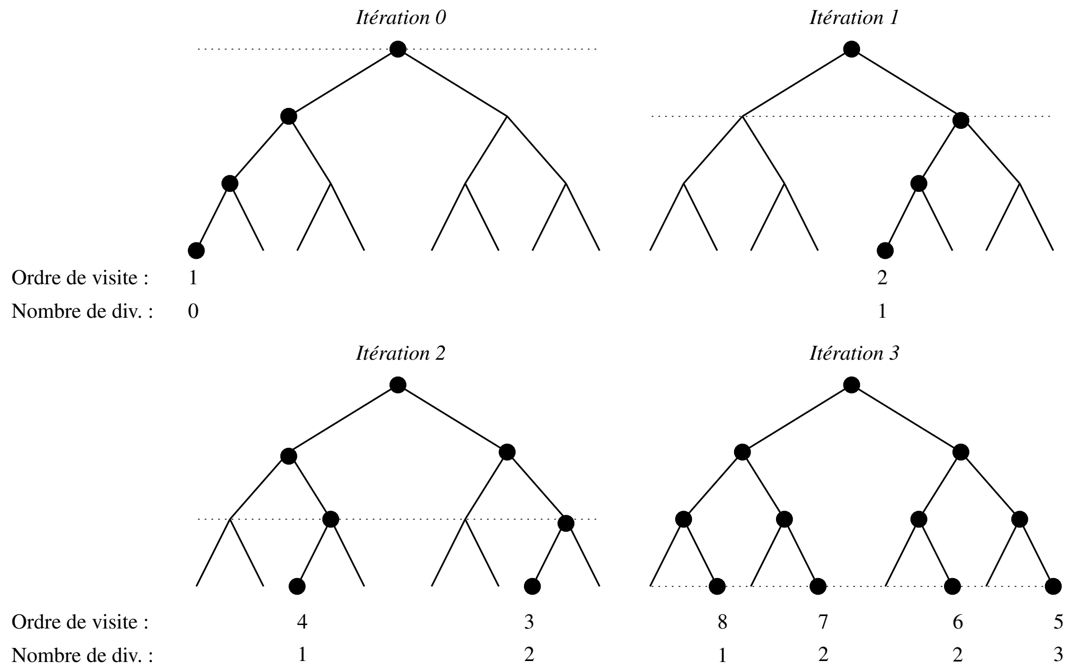


Figure 2.5 – Trace d'exécution de l'algorithme DDS

avec un nombre inférieur de divergences apparaissent.

2.5 Interleaved Depth-First Search (IDFS)

La méthode IDFS [Meseguer 1998], ou *recherche en profondeur d'abord entrelacée*, n'utilise pas explicitement la notion de divergence. Elle s'en inspire toutefois pour faire un parcours très particulier de l'espace de recherche qui, tout en suivant l'heuristique et s'en écartant petit à petit, balaye simultanément différents sous-arbres. Cette méthode a été conçue dans l'objectif de trouver le juste équilibre entre LDS et la méthode de recherche en profondeur classique (DFS : *Depth First Search*), les auteurs affublant la première d'un défaut de "dilettantisme" et la seconde de celui de "diligence excessive". IDFS se propose de pallier ces faiblesses dans une méthode complète. Comparé aux méthodes à base de divergences, cette amélioration a la particularité de ne pas faire plusieurs itérations et d'être concentrée sur une unique itération qu'elle gère en alternant le parcours de différents sous-arbres. Deux variantes ont été proposées pour cette méthode :

– *Pure_IDFS*, qui considère tous les sous-arbres parallèlement. Elle change de sous-

arbre à chaque fois qu'une feuille est atteinte. Cette méthode s'avère coûteuse et nécessite un espace mémoire exponentiel en la profondeur de l'arbre considéré. Elle est présentée comme la base théorique de sa variante pratique *Limited_IDFS*. L'algorithme 17 détaille son principe.

– *Limited_IDFS*, qui se concentre sur certains sous-arbres dits actifs pour les parcourir en parallèle à la manière de *Pure_IDFS*. Les autres sous-arbres sont traités séquentiellement à la manière de DFS. Les algorithmes 18 et 19 détaillent son principe.

La figure 2.6 présente la trace d'exécution des deux algorithmes sur un arbre ternaire qui compte 3 variables. Pour *Pure_IDFS*, la recherche se fait en alternant les différents sous-arbres du niveau 1 de racines respectives A, B et C à chaque fois qu'une feuille est rencontrée. Une fois dans l'un de ces sous-arbres, ses propres sous-arbres sont explorés alternativement de la même manière. Par exemple, D, E et F, les successeurs de A, sont alternés dès qu'une feuille est atteinte et ceci à chaque fois qu'on revisite le sous-arbre A. A chaque fois que *Pure_IDFS* passe d'un sous-arbre à un autre, une sauvegarde s'impose. Ceci donne l'ordre des visites décrit sur la figure. Pour *Limited_IDFS*, qui se veut plus pratique, on distingue deux types de niveaux, les niveaux parallèles et les niveaux séquentiels. Sur notre exemple, on suppose que le niveau parallèle commence au niveau 1. Les alternances se font comme dans *Pure_IDFS* mais entre un nombre limité de sous-arbres : les sous-arbres actifs. Les autres sous-arbres sont parcourus comme par DFS. Pour notre exemple, les sous-arbres de racines A et B sont considérés comme sous-arbres actifs et sont traités en premier. Une fois le parcours de l'arbre A terminé, on le remplace par le sous-arbre de racine C.

Algorithme 17 : Pure_IDFS(X, D, C)

1 **si** *InstanciationComplète(s)* **alors**
2   **retourner** succès ou échec
3 **si** *s n'est pas exploré* **alors**
4   générer successeurs(s)
5 **si** *successeurs(s)* $\neq \emptyset$ **alors**
6     $s' \leftarrow$ premier(successeurs(s))
7   Sol $\leftarrow$ Pure_IDFS(s',Sol,X,D,C)
8   **cas où** *Sol=continue % une solution est trouvée*
9   ajouter-dernier(successeurs(s),s')
10   **cas où** *Sol=succès % pas de solution ; arbre exploré*
11   **retourner** Sol
12   **cas où** *Sol=échec % pas de solution ; arbre non entièrement exploré*
13   ne rien faire
14 **si** *successeurs(s)* $= \emptyset$ **alors**
15   **retourner** échec
16 **sinon**
17   **si** *s est le nœud initial* **alors**
18   goto 5
19 **sinon**
20   **retourner** continue

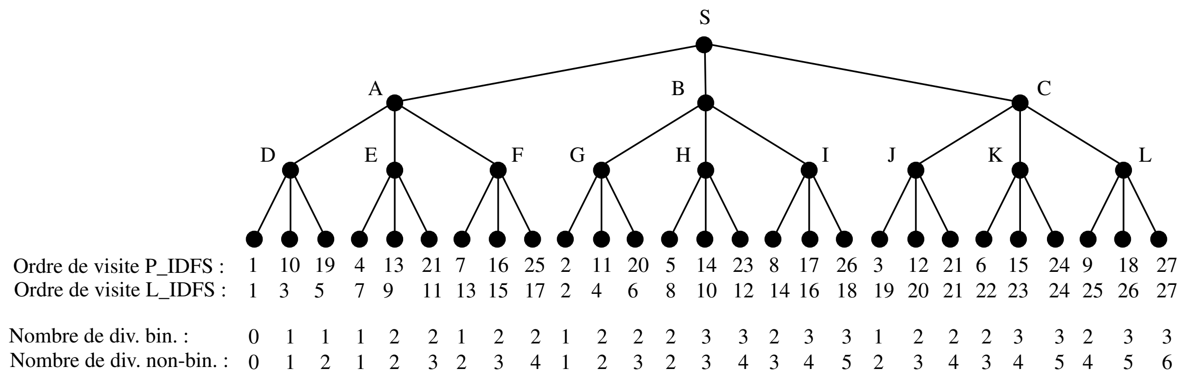


Figure 2.6 – Trace d'exécution de l'algorithme IDFS. La première ligne correspond à l'ordre des visites de Pure_IDFS ; la deuxième ligne correspond à l'ordre des visites de Limited_IDFS ; la ligne trois, respectivement quatre, correspond aux nombres de divergences sur la branche selon un comptage binaire, respectivement non-binaire.

Algorithme 18 : Limited_IDFS-itération(s, Sol, X, D, C)

```
1  si InstanciationComplète(s) alors
2  |  retourner succès ou échec
3  si s n'est pas exploré alors
4  |  générer successeurs(s)
5  si successeurs(s) ≠ ∅ alors
6  |  s' ← premier(successeurs(s))
7  |  Sol ← Limited_IDFS(s',Sol,X,D,C)
8  |  cas où Sol=continue
9  |  |  ajouter-dernier(successeurs(s),s')
10 |  cas où Sol=succès
11 |  |  retourner Sol
12 |  cas où Sol=échec
13 |  |  ne rien faire
14 si successeurs(s) = ∅ alors
15 |  retourner échec
16 sinon
17 |  retourner continue
```

Algorithme 19 : Limited_IDFS(X, D, C)

```
1  si s n'est pas exploré alors
2      générer successeurs(s)
3      active(s) ← k-premier(successeurs(s))
4  si active(s) ≠ ∅ alors
5      s' ← premier(active(s))
6      si niveau(s')+1 est un niveau parallèle alors
7          Sol ← Limited_IDFS(s',Sol,X,D,C)
8      sinon
9          Sol ← Limited_IDFS-itération(s',Sol,X,D,C)
10     cas où Sol=continue
11         ajouter-dernier(active(s),s')
12     cas où Sol=succès
13         retourner Sol
14     cas où Sol=échec
15         si successeurs(s) ne ∅ alors
16             ajouter-dernier(active(s),premier(successeurs(s)))
17     si active(s) = ∅ alors
18         retourner échec
19     sinon
20         si s est le nœud initial alors
21             goto 3
22     sinon
23         retourner continue
```

2.6 Reverse LDS (RLDS)

La méthode RLDS ou *recherche à divergence limitée inversée* [Prcovic 2002] est une variante de ILDS qui se distingue de cette dernière par l'ordre dans lequel on choisit les valeurs à examiner en premier : dans un arbre binaire, le fils droit est examiné avant le fils gauche. En général, cette méthode visite les feuilles de l'arbre ayant le moins de divergences d'abord et en cas d'égalité de divergences, elle visite celle qui serait le plus à droite dans l'arbre s'il était développé par DFS. La méthode RLDS correspond donc à la méthode ILDS dans laquelle les divergences seraient appliquées en haut de l'arborescence d'abord. Les algorithmes 20 et 21 montrent cette différence et la figure 2.7 détaille la trace d'exécution sur un arbre binaire de profondeur 3. Dans le cas d'un arbre binaire, RLDS nécessite au pire 2^n itérations où n est le nombre de variables.

Algorithme 20 : RLDS-itération(nœud, k, profondeur, Sol, X, D, C)

1. **si** *InstanciationComplète(Sol)* **alors**
2.   **retourner** Sol
3. s ← Successeurs (nœud)
4. **si** $s = \emptyset$ **alors**
5.   **retourner** $\emptyset$
6. Sol ← $\emptyset$
7. **si** $k > 0$ **alors**
8.   Sol ← RLDS-itération(fils-droit(s),k-1,profondeur-1, Sol, X, D, C)
9. **si** $Sol = \emptyset$ *et profondeur* $> k$ **alors**
10.   Sol ← RLDS-itération(fils-gauche(s),k,profondeur-1, Sol, X, D, C)
11. **retourner** Sol

Algorithme 21 : RLDS(X, D, C)

1. **pour** x ← *0 jusqu'à Profondeur-max* **faire**
2.   Sol ← RLDS-itération(nœud,x,Profondeur-max)
3.   **si** *InstanciationComplète(Sol)* **alors**
4.   **retourner** Sol
5. **retourner** $\emptyset$

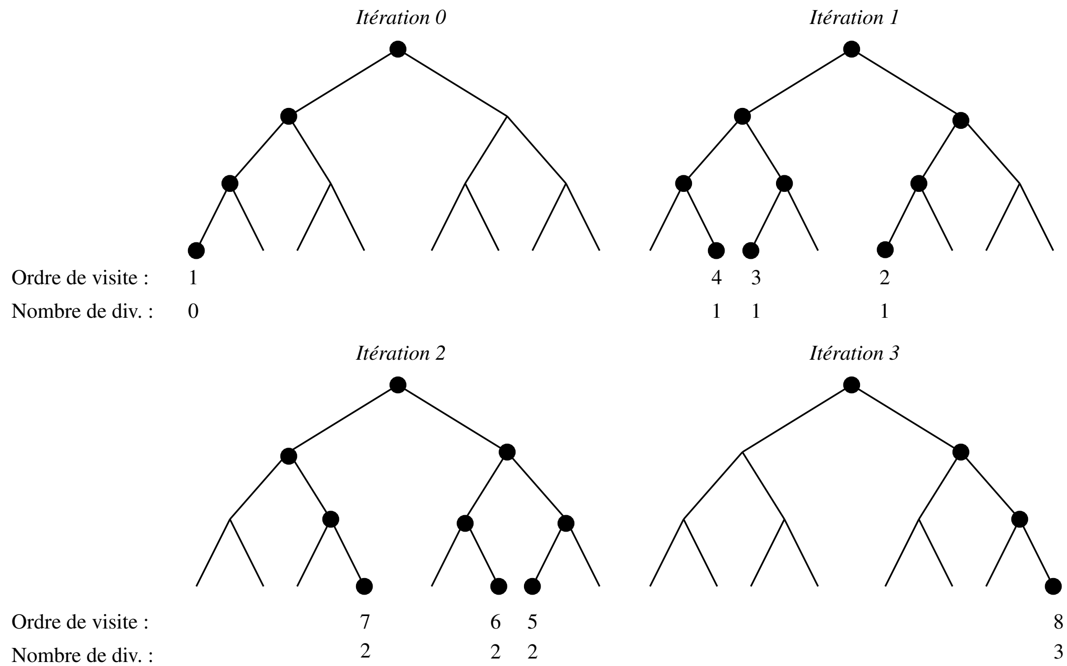


Figure 2.7 – Trace d'exécution de l'algorithme RLDS

2.7 Depth-Weighted Discrepancy Search (DWDS)

La méthode DWDS ou *recherche à divergence pondérée par profondeur* [Prcovic 2002] est une autre variante de LDS. Plutôt que de limiter le nombre de divergences, cette méthode limite la somme des profondeurs de divergences. En se fixant une valeur maximale pour la somme de profondeur de divergences, cette méthode examine différentes combinaisons de divergences. Ces combinaisons sont choisies de telle sorte que plus les divergences sont profondes, moins elles sont nombreuses.

Les algorithmes 22 et 23 détaillent le principe de cette méthode. Sur ces algorithmes, *decomposable(S,k,n)* vérifie si l'entier S est décomposable en entiers de l'intervalle $[k,n]$, chose que l'on peut vérifier en testant si S, k, et n vérifient deux inéquations proposées dans [Prcovic 2002].

Dans le cas d'un arbre binaire, DWDS nécessite au pire $n*(n+1)/2+1$ itérations où n représente le nombre de variables. Cependant, la redondance dans la génération des nœuds internes est supérieure à celle de LDS. Quoiqu'adaptable à des arbres non-binaires, l'intérêt de cette méthode reste très lié aux arbres binaires. En cas de généra-

lisation, la profondeur d'un nœud n'est plus suffisante pour prévoir les divergences qui restent possibles.

Algorithme 22 : DWDS-itération(nœud, s, profondeur, Sol, X, D, C)

```
1  si InstanciationComplète(Sol) alors
2  │ retourner Sol
3  f ← Successeurs (nœud)
4  si f = ∅ alors
5  │ retourner ∅
6  si décomposable(s,profondeur+1,n) alors
7  │ retourner DWDS-itération(premier(f),s,profondeur+1,Sol,X,D,C)
8  tant que f ≠ ∅ faire
9  │ si décomposable(s-profondeur,profondeur+1,n) alors
10 │ │ retourner
   │ │ DWDS-itération(premier(f),s-profondeur,profondeur+1,Sol,X,D,C)
11 │ f ← f \ premier(f)
12 retourner ∅
```

Algorithme 23 : DWDS(X, D, C)

```
1  pour x ← 0 jusqu'à n.(n+1)/2 faire
2  │ Sol ← DWDS-itération(nœud,x,0,Sol,X,D,C)
3  │ si InstanciationComplète(Sol) alors
4  │ │ retourner Sol
5  retourner ∅
```

2.8 Discrepancy-Bounded Depth First Search (DBDFS)

La méthode DBDFS ou *recherche à divergence bornée* proposée par Beck et Perron [Beck 2000] suit un schéma classique de recherche en profondeur d'abord, mais limite les solutions obtenues par le nombre de divergences autorisé. Ainsi, dans une itération i donnée, on ne cherche que les feuilles correspondant à un nombre de divergences entre les deux bornes ik et $k(i+1)-1$ où k est un paramètre traduisant la largeur de la recherche. L'objectif de cette méthode est de minimiser les redondances tout en préservant

l'ordre de visite des solutions potentielles de OLDS. Cette méthode a été testée efficacement sur des problèmes d'ordonnancement de type job shop. L'algorithme DBDFS suit le même principe que l'algorithme ILDS avec le nombre de divergences qui progresse par intervalle au lieu d'être incrémenté à chaque itération. Sur la figure 2.8, nous pouvons observer la trace d'exécution de cet algorithme sur un arbre binaire comptant 4 variables et avec une largeur de recherche égale à 2. La première itération est consacrée aux feuilles à 0 et 1 divergence et la deuxième aux feuilles à 2 et 3 divergences. La dernière itération ne contient qu'une feuille à 4 divergences, étant donné que c'est le nombre maximum de divergences possible pour cet arbre.

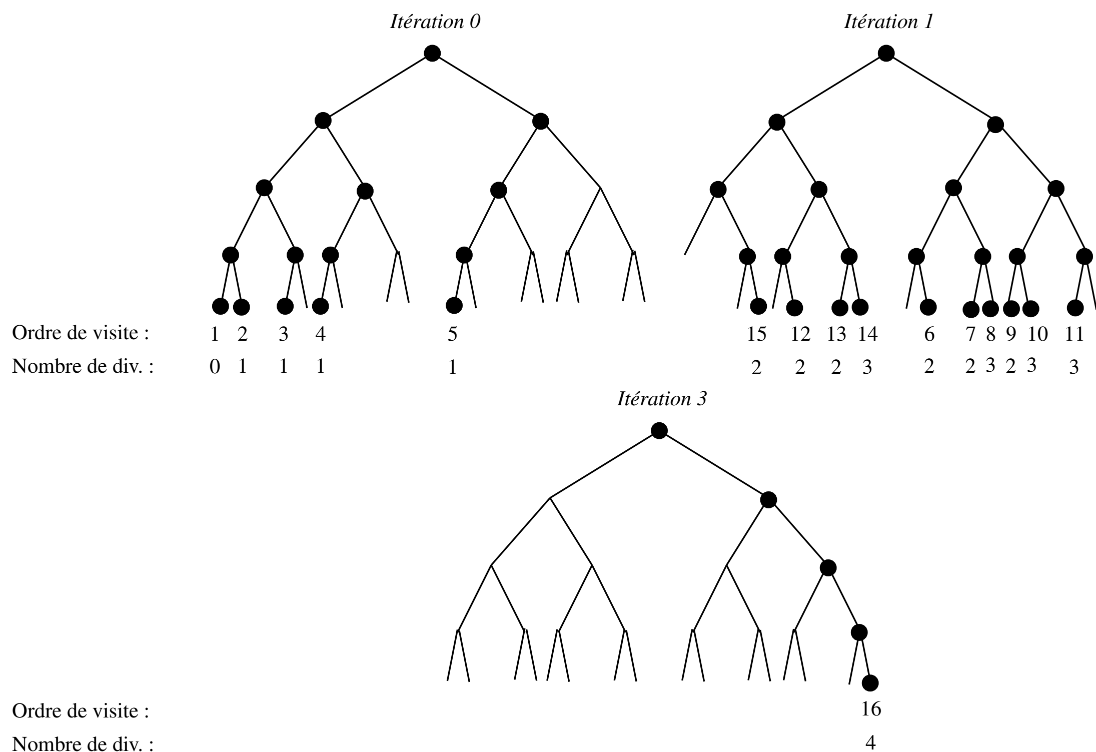


Figure 2.8 – Trace d'exécution de l'algorithme DBDFS (largeur=2)

DBDFS peut être intégrée à la plupart des schémas de recherche à base de divergences présentés. Son intégration à DWDS peut être considérée comme une généralisation de DFS. Par exemple, dans le cas d'une largeur de $n*(n+1)/2$ avec n le nombre de variables binaires, on obtient de la combinaison de DWDS et DBDFS une trace équiva-

lente à celle du parcours de DFS et dans le cas d'une largeur de 1, une trace d'exécution équivalente à celle de DWDS.

2.9 Conclusion

Dans ce chapitre, nous présentons un état de l'art de différentes méthodes de résolution à base de divergences. Cette étude nous a permis de constater que ces méthodes présentent des particularités qui permettent un parcours atypique par divers aspects de l'espace de recherche. On peut noter que les méthodes à base de divergences ont été peu employées jusqu'ici particulièrement pour la résolution de CSP.

La recherche à divergences étant considérée comme un principe de recherche général, on peut facilement imaginer l'adaptation et la conception de mécanismes de manière à pouvoir, tout en étant dans le cadre du paradigme divergence, concurrencer les résultats obtenus par les meilleurs méthodes de résolution connues actuellement et qui ne sont pas à base de divergences.

La raison d'être de ce survol de l'état de l'art en matière de méthodes à divergences est de fournir un socle pour construire et présenter les propositions que nous allons faire dans le chapitre suivant.

CHAPITRE 3

HEURISTIQUES D'INSTANCIATION ET GESTION DES DIVERGENCS POUR LA RÉSOLUTION DE CSP

DAns la première partie de ce chapitre, nous proposons des techniques permettant d'améliorer les méthodes de recherche générales en exploitant les échecs rencontrés pendant la résolution. D'autres techniques spécifiques aux méthodes à base de divergences sont ensuite proposées.
Dans la deuxième partie, nous présentons les expérimentations menées pour attester de l'efficacité de nos propositions sur des benchmarks.

Certains des résultats proposés dans ce chapitre ont été présentés dans la conférence internationale CP-AI-OR'07 [Karoui 2007b], les JFPC'05 [Karoui 2005], ainsi que les ROADEF'06 [Karoui 2006], ROADEF'07 [Karoui 2007a] et ROADEF'09 [Karoui 2009].

3.1 Introduction

Comme mentionné dans l'état de l'art, la résolution d'un CSP se base sur différentes composantes comme la recherche arborescente, les techniques de propagation, les heuristiques d'instanciation (choix des variables et des valeurs), etc. Ainsi, l'amélioration d'une méthode de résolution passe par l'amélioration d'une de ces composantes et de son bon agencement dans l'ensemble pour des interactions bénéfiques dans la méthode composée.

Dans ce chapitre, nous proposons quelques techniques pouvant contribuer à l'amélioration des méthodes générales de résolution de CSP, ainsi que des techniques dédiées aux méthodes basées sur les divergences.

L'intégration des techniques de propagation dans des méthodes à base de divergences adopte les mêmes hypothèses que dans le chapitre précédent. Ceci implique que dans la suite, lorsqu'une ou plusieurs variables sont éliminées par propagation suite aux précédentes instanciations, nous ne comptons plus ces valeurs et le comptage des divergences fonctionne comme si ces valeurs n'existaient pas. Bien entendu, quand on remet en question l'instanciation qui les a éliminées, ces valeurs sont restaurées et rentrent donc de nouveau dans les calculs de divergences.

3.2 Mécanismes d'amélioration pour les méthodes de recherche arborescente

Les mécanismes proposés dans ce paragraphe sont susceptibles d'être utilisés dans n'importe quel type de méthode de recherche arborescente.

Lors du parcours de l'espace de recherche, une méthode de résolution trouve rarement une solution admissible dès la première tentative. En général, une méthode, même dotée de bonnes heuristiques, échoue plusieurs fois avant d'arriver à une feuille correspondant à une solution admissible, *i.e.* respectant l'ensemble des contraintes du problème. Les échecs rencontrés pendant la résolution constituent des informations que l'on peut rajouter aux données du problème afin de les exploiter pour guider la recherche.

Dans les paragraphes suivants, nous détaillons les diverses manières selon lesquelles ces échecs peuvent être exploités. On désignera par heuristique la méthode permettant de

sélectionner la variable ou la valeur à considérer en priorité lors du parcours de l'espace de recherche.

3.2.1 Pondération des variables

Nous proposons de mémoriser les échecs rencontrés lors de la résolution via des pondérations associées aux variables du problème. Nous notons par la suite $W_{var}(i)$ le poids de la variable X_i. Tous les poids des variables sont initialement identiques. Lors de la résolution, le poids d'une variable est incrémenté à chaque fois que cette variable est impliquée dans un échec, c'est-à-dire lorsque l'apparition d'un domaine vide pour cette variable est directement responsable d'un échec. Ainsi, une variable ayant un poids élévé correspond à une variable pour laquelle de nombreux échecs auront été rencontrés lors de la résolution. Ces informations de pondération des variables peuvent être utilisées seules afin de définir une heuristique dynamique d'ordre sur les variables (noté `Wvar` par la suite) afin de considérer de manière prioritaire les variables sources d'échecs lors de précédentes itérations de la recherche.

Le principe général de pondération des variables proposé ici peut également être combiné dans n'importe quelle heuristique dynamique d'ordre sur les variables. Par exemple, nous pouvons définir l'heuristique dynamique `dom⊕Wvar` consistant à sélectionner en priorité la variable ayant le plus petit domaine et en cas d'égalité la variable ayant rencontré le plus d'échecs. Nous pouvons également définir l'heuristique `dom/Wvar` permettant de sélectionner la variable ayant à la fois le plus petit domaine et la plus grande pondération.

3.2.2 Exemple illustratif

La figure 3.1 illustre le principe de l'heuristique d'instanciation `Wvar` au sein d'une procédure de backtrack chronologique. Considérons un CSP composé de quatre variables X_1, X_2, X_3 et X_4 de domaines D_1, D_2, D_3 et D_4, reliées par des contraintes comme l'indique la partie gauche de la figure. On ne détaille pas la nature de ces contraintes. La partie droite de la figure décrit comment le poids d'une variable est incrémenté quand

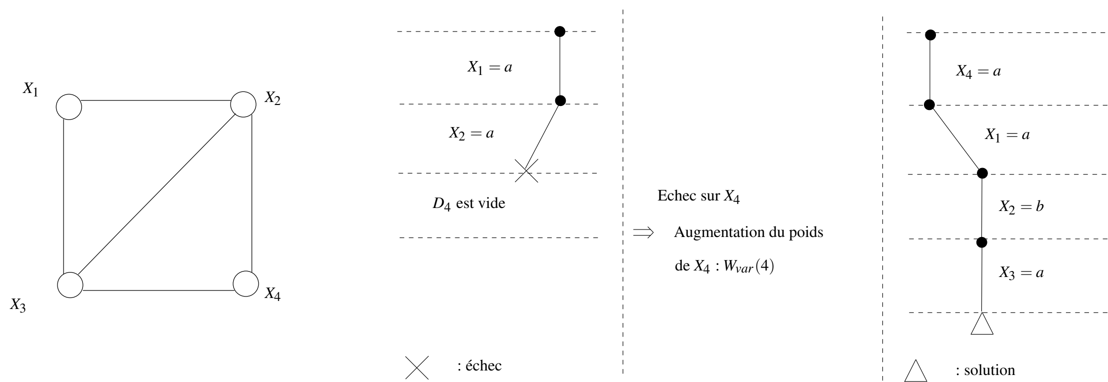


Figure 3.1 – Heuristique de poids sur les variables

un échec est rencontré. Supposons que l'heuristique d'ordre sur les variables suit l'ordre lexicographique : l'heuristique choisit X_1 ensuite X_2. Supposons qu'à la suite de ces instanciations, le domaine de X_4 devienne vide. Le poids de X_4 est alors incrémenté. Lors d'un backtrack, nous considérons ainsi en premier l'instanciation de X_4 puisque son poids est plus important que celui des autres variables, puis l'heuristique considère X_1 X_2 et enfin X_3 afin d'aboutir sur cet exemple à une solution.

3.2.3 Pondération des valeurs

A l'instar des variables, nous proposons dans cette partie de mémoriser les échecs rencontrés lors de la résolution via des pondérations associées aux valeurs des différentes variables. Par la suite, le poids de la valeur v_i de la variable X_j est noté $W_{val}(i,j)$. Tous les poids de valeurs sont initialement identiques. Plusieurs techniques peuvent être envisagées pour incrémenter ces poids, nous en présentons trois ci-dessous.

La première technique d'incrémentation notée `Wval1` augmente le poids d'une valeur si celle-ci fait partie de la dernière instanciation avant que la résolution ne rencontre un domaine vide. Cette instanciation a été la dernière à être réalisée et c'est donc celle qui, par propagation, a vidé le domaine d'une autre variable, ce qui a engendré l'échec détecté.

La deuxième technique d'incrémentation notée `Wval2` augmente les poids des valeurs de la variable accusant un domaine vide qui ont été filtrées par les propagations

de la dernière instanciation. Ces valeurs sont nécessaires à la solution. Donc leurs variables devraient être traitées au préalable, en priorité, de telle sorte qu'elles ne soient pas supprimées car leurs éliminations peut nous conduire à un échec.

La troisième technique d'incrémentation `Wval3` augmente les poids des valeurs de la variable accusant un domaine vide, qui ont été filtrées par des propagations des instanciations autres que la dernière instanciation. Si ces valeurs n'avaient pas été supprimées et que leurs variables avaient été considérées au préalable, on aurait pu éviter l'échec. Le fait de les privilégier permet d'éviter cette situation et d'avancer dans l'exploration de l'espace de recherche.

3.2.4 Exemple illustratif

Considérons un CSP composé de quatre variables X_1, X_2, X_3 et X_4 de domaines D_1, D_2, D_3 et D_4, reliées par des contraintes que nous ne détaillons pas. Supposons que nous considérons comme méthode de recherche un backtrack chronologique avec comme heuristique d'ordre sur les variables l'ordre lexicographique. Dans la figure 3.2, un échec apparaît sur la variable X_4 après instanciation des variables X_1, X_2 et X_3 : le domaine D_4 initialement égal à $\{v_1, v_2, v_3, v_4, v_5\}$ est devenu vide. Pour illustrer les différentes techniques d'incrémentation du poids des valeurs, nous supposons que $(X_4 = v_1)$ et $(X_4 = v_2)$ sont filtrées par l'instanciation de X_3 : $(X_3 = v_4)$ et que les autres valeurs de X_4 sont filtrées par les instanciations de X_1 et X_2. Trois cas de figure sont envisageables, correspondant aux trois techniques d'incrémentation sur les valeurs étudiées :

- **Cas `Wval1`)** Le poids de la dernière instanciation (dans notre exemple, valeur v_4 de la variable X_3 : $W_{val}(3,4)$) est incrémenté.
- **Cas `Wval2`)** Les poids des instanciations qui ont été filtrées par la dernière instanciation (celle de la variable X_3), c'est-à-dire $W_{val}(4,1)$ et $W_{val}(4,2)$, sont incrémentés.
- **Cas `Wval3`)** Les poids des instanciations qui ont été filtrées par des instanciations autres que la dernière instanciation, c'est-à-dire $W_{val}(4,3)$, $W_{val}(4,4)$ et $W_{val}(4,5)$, sont incrémentés.

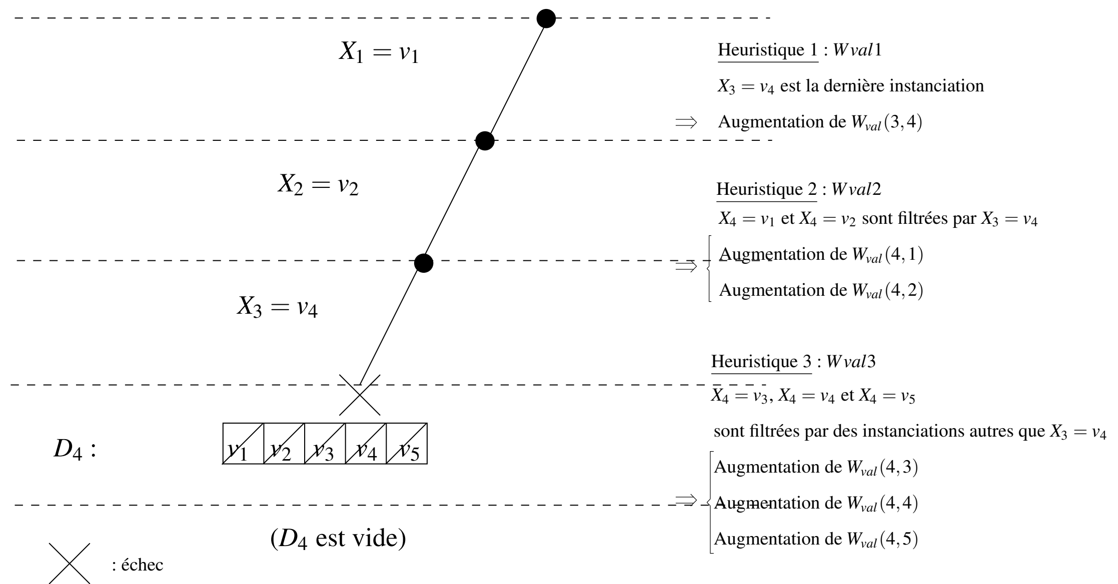


Figure 3.2 – Pondération des valeurs

Ces différentes techniques de pondération des valeurs peuvent être utilisées seules comme heuristique dynamique d'ordre sur les valeurs ou combinées avec d'autres heuristiques dynamiques sur les valeurs.

3.3 Mécanismes d'amélioration pour les méthodes de recherche à divergences

Les méthodes à divergences explorent l'espace de recherche de manière particulière. Etudier les particularités de ce parcours, notamment l'aspect divergence autour duquel est bâti le principe, et le problème des redondances dues aux nombreuses itérations, peut se révéler bénéfique pour proposer des variantes plus performantes. Ce paragraphe s'inscrit dans ce cadre. Il propose des mécanismes spécifiques aux méthodes de recherche arborescente basées sur le concept de divergences.

3.3.1 Restriction des divergences

Dans LDS, le nombre de divergences est incrémenté jusqu'à ce que le maximum de divergences autorisées par la recherche soit atteint. Pour éviter certaines itérations inutiles, nous pouvons vérifier à chaque étape si la recherche a besoin d'une divergence

supplémentaire ou pas avant de poursuivre la recherche. Si c'est inutile, et que la dernière itération a été interrompue pour une autre raison qu'une limite sur les divergences (une incohérence a été détectée), l'exploration peut être stoppée : le problème étudié n'admet alors pas de solution. Supposons que *CurrentMaxDiscr* est le nombre de divergences maximal autorisé dans une itération donnée et que *UsedDiscr* est le nombre de divergences utilisées. L'algorithme 24 décrit le mécanisme de restriction des divergences. De cette manière, nous pouvons obtenir des méthodes qui incrémentent le nombre de divergences autorisé en fonction des besoins réels du problème traité et, par suite, font moins d'itérations que les méthodes plus classiques de la famille LDS.

DRestrict (algorithme 24) est la procédure qui teste la restriction des divergences. Elle décide s'il est nécessaire ou non de diverger. Elle peut être appelée dans la méthode LDS ou une de ses variantes pour mettre un terme aux redondances occasionnées lors de la tentative de résolution de problèmes insolubles.

Algorithme 24 : DRestrict(CurrentMaxDiscr,UsedDiscr)

1 $c \leftarrow (CurrentMaxDiscr - UsedDiscr)$
2 **si** $c \geq 0$ % plus de divergences possibles que de divergences utiles pour l'itération **alors**
3 | **retourner** true % il est inutile de continuer la recherche (pas de solution)
4 **sinon**
5 | **retourner** false % la recherche continue

3.3.1.1 Exemple illustratif

Soit le CSP insoluble suivant décrit par son graphe d'incohérence (figure 3.3). Sa résolution avec le mécanisme de restriction s'arrête à la quatrième itération vu qu'il y a uniquement trois divergences de consommées alors que quatre sont permises. L'itération échoue à cause des domaines vidés sans utiliser la borne des divergences. La résolution du même CSP par LDS continue jusqu'à la neuvième itération pour générer des arbres de recherche identiques et donc augmenter la redondance pour un problème que l'on peut décider insoluble dès la quatrième itération. Etant donné que la condition d'arrêt n'est pas liée au nombre de divergences, le parcours de la dernière itération peut être

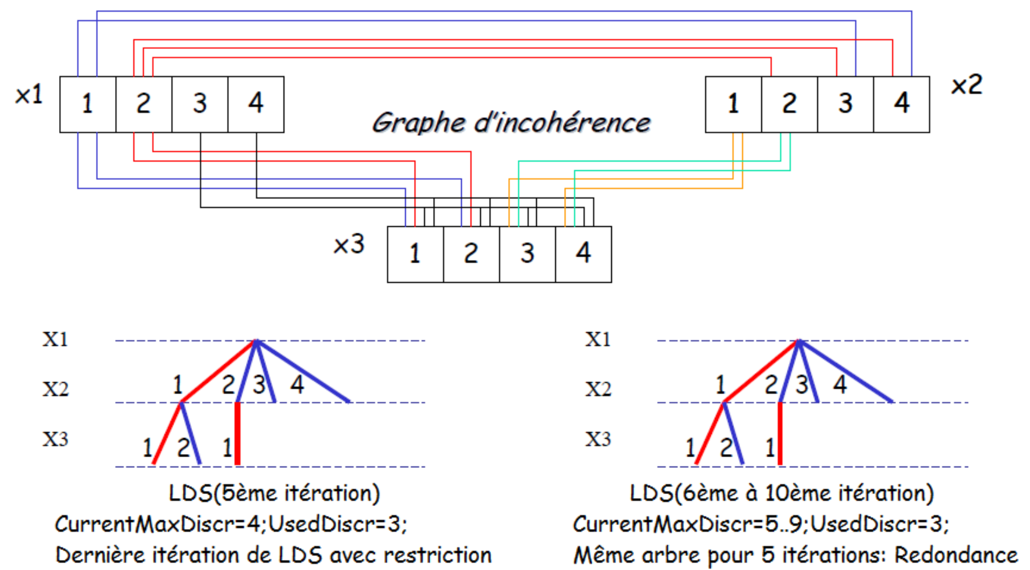


Figure 3.3 – Restriction des divergences

assimilé à celui d'une autre méthode de résolution non itérative (CB par exemple). De cette manière, on est certain que c'est une méthode complète : on n'a pas raté la solution, elle n'existe pas.

Dans cet exemple, l'itération à laquelle s'arrête la version qui profite des restrictions est associée à *CurrentMaxDiscr*$=4$. A la fin de cette itération les domaines sont vides, nous n'avons plus de valeurs pour diverger, et *UsedDiscr* est égal à 3. Une divergence serait encore possible, mais une itération supplémentaire est inutile. Il est certain que si l'on passe aux prochaines itérations comme le cas de LDS sans restrictions, l'arbre de recherche va être identique à l'arbre courant pour toutes les prochaines itérations. La limite sur les divergences n'est pas responsable de l'interruption de la recherche. Il n'existe pas de solution pour ce problème.

3.3.2 Différents modes de comptage des divergences

Après le mécanisme de restriction sur les divergences, nous présentons ici des éléments sur le comptage de ces divergences qui, s'il ne présente pas d'ambiguité pour les

arbres binaires, peut être réalisé de diverses manières pour les arbres non-binaires.

3.3.2.1 Comptages binaire et non-binaire

Pour les arbres non-binaires, deux modes de comptage peuvent être adoptés.

- Le mode binaire : explorer la branche associée à la meilleure valeur, selon l'heuristique d'ordre sur les valeurs, n'engendre pas de divergences alors que l'exploration des autres branches liées à la même variable mais ne correspondant pas au meilleur choix de l'heuristique implique à chaque fois une unique divergence.
- Le mode non-binaire : les valeurs sont rangées par l'heuristique d'ordre sur les valeurs de telle sorte que la meilleure valeur a le premier rang ; explorer la branche associée à la valeur de rang $k > 1$ implique $k-1$ divergences : plus on s'écarte du meilleur choix de l'heuristique, plus le nombre de divergences augmente.

La figure 3.4 illustre le cas de 3 variables $X_{i\ (i=1..3)}$ avec $|D_i| = 3, \forall i$. La première ligne correspond au nombre de divergences en comptage non-binaire. La seconde ligne correspond au nombre de divergences en comptage binaire.

Pour un CSP à variables entières, le principe LDS peut être associé à un comptage binaire ou non-binaire [Karoui 2005, Levasseur 2007, Gacias 2008]. Notons que le comptage binaire correspond à une recherche à grosse granularité alors que le comptage non-binaire correspond à une recherche à granularité fine, suivant une terminologie classique dans les méthodes de recherche.

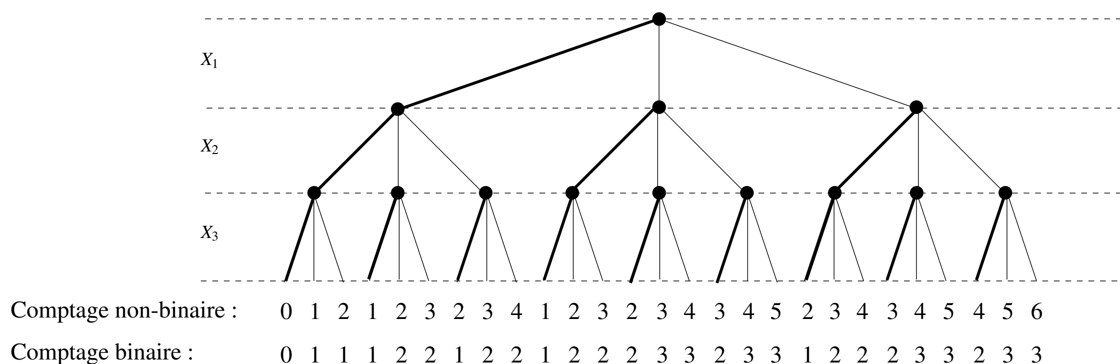


Figure 3.4 – Modes binaire et non-binaire pour le comptage des divergences

Une divergence de plus en comptage binaire correspond à l'exploration de toutes les valeurs d'une variable sur laquelle on n'a pas divergé jusque-là. Si les variables du CSP comptent d valeurs chacune, on obtient d branches supplémentaires.

Une divergence de plus en comptage non-binaire correspond à l'exploration d'une seule valeur supplémentaire au plus d'une variable.

3.3.2.2 Comptage mixte

Nous pouvons envisager de combiner le comptage binaire et le comptage non-binaire pour adopter un comptage mixte [Karoui 2005, Levasseur 2007, Gacias 2008] [Karoui 2009]. L'idée est de concentrer la recherche sur certains niveaux de l'arbre de recherche et donc de parcourir les variables de ces niveaux par une recherche à granularité fine lorsqu'on prévoit que les instanciations de ces variables ont un grand impact sur la recherche. Les autres niveaux sont alors considérés comme moins importants et leurs variables peuvent donc être parcourues par une recherche à forte granularité. Pour mélanger les deux modes de comptage, nous avons besoin de certains paramètres comme la profondeur de l'arbre de recherche. Les deux parties de l'arbre peuvent être explorées différemment : comptage binaire dans la partie haute, non-binaire dans la partie basse, ou inversement. Pour ajuster de tels paramètres, on peut s'inspirer de mécanismes d'apprentissage. Par exemple, nous pouvons nous baser sur le nombre d'échecs rencontrés par une variable donnée pour choisir la partie de l'arbre dans laquelle elle va être parcourue et donc la méthode de parcours associée.

3.3.3 Positionnement des divergences

Dans une méthode de recherche à divergences, il est possible de diverger prioritairement en haut de l'arbre, ou prioritairement en bas. En pratique, le choix du positionnement des divergences peut être responsable de grandes différences dans les parcours des espaces de recherche et améliorer voire détériorer les résultats. Malgré l'intuition qui consiste à dire que les divergences doivent prioritairement faites en haut de l'arbre, selon le problème étudié, il nous paraît judicieux de tester la stratégie permettant d'obtenir en

moyenne les meilleurs résultats.

3.4 Combinaison de mécanismes

Plusieurs combinaisons des améliorations proposées précédemment peuvent être considérées. Nous pouvons pour cela greffer les heuristiques sur les variables et les valeurs que nous avons présentées sur des méthodes de recherche de type Forward-Checking, MAC ou à base de divergences. Concernant les méthodes à divergences, nous pouvons également prendre en compte le mécanisme de restriction, différents types de comptage et différents emplacements des divergences sur ces méthodes à base de divergences.

3.4.1 YIELDS

La méthode Yet ImprovEd Limited Discrepancy Search (YIELDS) [Karoui 2007b] est un exemple de combinaison de quelques mécanismes d'amélioration présentés précédemment. Les algorithmes 25 et 26 décrivent globalement cette méthode. Le principe de YIELDS est exactement le même que LDS : initialement, on considère les branches de l'arbre qui cumulent le plus petit nombre de divergences. La première amélioration est que cette méthode profite du mécanisme de l'association des poids aux variables (cf. § 3.2.1) : un poids (initialement le même pour toutes les variables) est associé à chaque variable (algorithme 25, ligne 7). A chaque fois qu'une variable échoue, son poids est incrémenté pour guider les prochains choix de l'heuristique (algorithme 26, ligne 10). La seconde amélioration est qu'elle utilise le mécanisme de restriction des divergences (algorithme 25, ligne 11), ce qui signifie que le nombre de divergences n'est pas incrémenté automatiquement comme dans LDS. L'intégration de ces mécanismes fait que YIELDS consomme moins de divergences que les autres méthodes à base de divergences. Pour affiner la composition de la méthode que nous proposons, nous associons à YIELDS différents modes de comptage.

Dans *YIELDS_Framework* (*cf.* algorithme 25), *MaxDiscr* est le nombre maximal de divergences et la fonction *Increment* met à jour le nombre de divergences autorisées jusqu'à ce que, soit la valeur de *MaxDiscr* calculée en fonction du mode de comptage

Algorithme 25 : YIELDS_Framework(Sol, X, D, C)

1 $Sol \leftarrow \emptyset$
2 % Initialisation du nombre maximal global des divergences :
3 $MaxDiscr \leftarrow Init(X,D)$
4 % nombre de divergences maximal pour l'itération courante :
5 $CurrentMaxDiscr \leftarrow 0$
6 % Initialisation des vecteurs poids :
7 Init(W_{var},W_{val})
8 **tant que** *(Sol = ∅) et (CurrentMaxDiscr ≤ MaxDiscr)* **faire**
9 $Sol \leftarrow YIELDS_Iteration(X,W_{var},W_{val},D,C,CurrentMaxDiscr,Sol)$
$Update(UsedDiscr)$
10 $CurrentMaxDiscr \leftarrow Increment(CurrentMaxDiscr)$
11 **si** *DRestrict(CurrentMaxDiscr,UsedDiscr)* **alors**
12 **retourner**$\emptyset$
13 **retourner**Sol

Algorithme 26 : YIELDS_Iteration($Sol,X,D,C,W_{var},W_{val},CurrentMaxDiscr$)

1 **si** $X = \emptyset$ **alors**
2 **retourner**Sol % Toutes les variables sont instanciées
3 **sinon**
4 $X_i \leftarrow$VarOrder(X,W_{var})
5 $v_j \leftarrow$ValOrder$(D_i,W_{val},CurrentMaxDiscr)$
6 **si** *Exist(v_j)* **alors**
7 $(X',D',C') \leftarrow Propagate(X\backslash\{X_i\},D,C,(X_i,v_j))$
8 Sol ← YIELDS_Iteration($Sol \cup \{(X_i,v_j)\},X',D',C',W_{var},$
W_{val},UpdateDiscrepancy($CurrentMaxDiscr$))
9 **si** *InstanciationComplète(Sol)* **alors**
10 **retourner**Sol
11 **sinon**
12 **si** $CurrentMaxDiscr > 0$ **alors**
13 $D_i \leftarrow D_i\backslash\{v_j\}$
14 **retourner** YIELDS_iteration($Sol,X,D,C,W_{var},$
W_{val},UpdateDiscrepancy($CurrentMaxDiscr$))
15 **sinon**
16 UpdateWeights(W_{var},W_{val},X_i,v_j)
17 **sinon**
18 **retourner**$\emptyset$

adopté est atteinte, soit une solution est trouvée.

Dans *YIELDS_Iteration* (algorithme 26), chaque fois que *YIELDS_Iteration* est appelée, la fonction *UpdateDiscrepancy* met à jour *CurrentMaxDiscr* comme le mode de comptage le permet. La fonction *UpdateWeights* met à jour les poids des variables et/ou des valeurs, selon les mécanismes selectionnés. *VarOrder* fait référence à l'heuristique d'ordre sur les variables choisie et *ValOrder* fait référence à l'heuristique d'ordre sur les valeurs choisie.

3.4.2 Exemple illustratif

Soit un CSP composé de quatre variables X_1, X_2, X_3, et X_4. Chaque variable admet trois valeurs de telle sorte que $D_1 = \{1,2,3\}$, $D_2 = \{4,5,6\}$, $D_3 = \{7,8,9\}$, et $D_4 = \{10,11,12\}$. Les contraintes du CSP sont décrites par l'ensemble des couples d'instanciations incompatibles : $C = \{\{(X_1,1),(X2{,}5)\},\{(X_1,2),(X_1,5)\},\{(X_1,3),(X_2,4)\}, \{(X_2,4),(X_3,8)\},\{(X_2,4),(X_4,12)\},\{(X_2,5),(X_4,12)\},\{(X_2,6),(X_3,8)\},\{(X_2,6),(X_4,12)\},\{(X_3,7),(X_4,10)\},\{(X_3,7),(X_4,11)\},\{(X_3,8),(X_4,11)\},\{(X_3,9),(X_4,10)\},\{(X_3,9),(X_4,11)\}\}$.

Nous pouvons observer sur cet exemple que la résolution par YIELDS (figure 3.6) est accélérée en termes de nombre de nœuds développés par rapport à la résolution par LDS (figure 3.5).

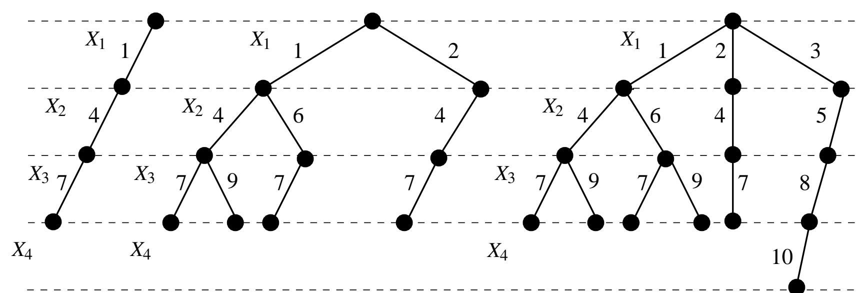


Figure 3.5 – Arbre de recherche développé par LDS (29 nœuds)

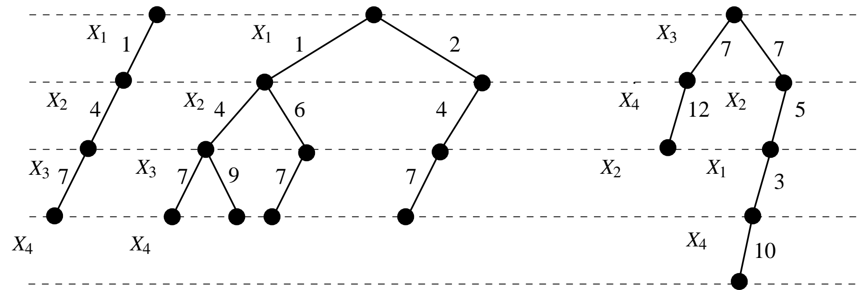


Figure 3.6 – Arbre de recherche développé par YIELDS (21 nœuds)

3.4.3 Exploitation des no-goods

Nous présentons ici l'intégration d'un mécanisme d'apprentissage à la méthode LDS basé sur une mémorisation des no-goods. Un nogood est un ensemble d'instanciations incompatibles qui mènent à un échec partiel. Comme on ne peut se permettre de tout mémoriser pour des problèmes de grande taille, nous ne mémorisons que les no-goods plus récemment apparus. Pour remédier au fait de ne pouvoir se permettre une mémorisation exhaustive, nous proposons d'associer des poids aux instanciations associées à ces no-goods. Les poids considérés jusqu'à maintenant, que ce soit en association avec des variables ou des valeurs, ont des valeurs entières. Dans ce paragraphe, nous passons à des poids associés à des valeurs réelles obtenues suite à l'analyse des no-goods.

Si l'on associe à chaque paire d'instanciations une valeur qui reflète leur taux d'incompatibilité, on peut traduire ces incompatibilités par des valeurs :

– 0 pour une compatibilité parfaite (instanciations compatibles) ;
– 1 pour une compatibilité inexistante (instanciations incompatibles).

Le problème initial contient des instanciations compatibles et des instanciations incompatibles ; une matrice binaire suffit donc pour le définir. On peut, suite à l'analyse des no-goods, dégager des valeurs intermédiaires entre 0 et 1 pour exploiter les informations que nous recueillons suite à chaque échec et guider la recherche dans les prochaines étapes. Ainsi, en plus d'avoir des couples d'instanciations compatibles et incompatibles, nous obtenons des couples d'instanciations probablement compatibles correspondants à des valeurs appartenant à $]0,1[$.

Partant du fait que deux CSP sont dits équivalents s'ils admettent le même ensemble de solutions, la déduction d'une incompatibilité de valeur 1 peut nous mener à un CSP non équivalent. L'incrémentation ne doit jamais atteindre la valeur 1 pour ne bannir aucune solution possible du problème.

Pour un tel traitement, on peut représenter le CSP par une matrice symétrique qui fait apparaître toutes les valeurs sur les lignes ainsi que sur les colonnes. Sur la case correspondant à deux instanciations de deux variables différentes, on note la relation qui les lie : compatibilité, incompatibilité ou probabilité de compatibilité déduite. Cette matrice évolue au fur et à mesure que la résolution avance. Ainsi, les valeurs de la matrice initialement à 0 (pour traduire la compatibilité) se transforment en fonction de probabilités déduites des no-goods. Plus une instanciation va apparaître parmi les no-goods, plus sa valeur se rapprochera de celle d'une incompatibilité. Ces informations déduites peuvent guider l'heuristique. Par exemple, considérant les nouvelles valeurs comme des conflits partiels (valeur entre 0 et 1), les prochains choix de l'heuristique `min-conflict` vont être influencés pour mieux choisir les instanciations à favoriser.

3.4.4 Exemple illustratif

Reprenons l'exemple du paragraphe 3.4.2 concernant un CSP à quatre variables ayant trois valeurs chacune. La matrice d'incompatibilité entre toute paire d'instanciation est décrite dans la figure 3.7. Dans une résolution par NG-YIELDS, la version de LDS qui intègre les mécanismes d'amélioration décrits précédemment (*cf.* exemple YIELDS) ainsi que l'exploitation des no-goods, ce CSP évolue suite à la détection du no-good (A,B,C) décrit sur la figure 3.9, de telle sorte qu'on obtienne la matrice de la figure 3.8 où les relations entre les couples d'instanciations compatibles avec probabilité sont marquées par des cercles grisés. Partant de cette matrice et considérant ces cercles comme des pseudo-conflits, la résolution par NG-YIELDS, comme on peut l'observer sur la figure 3.9, converge plus rapidement que si l'on s'était basé uniquement sur les données de base du problème considéré. YIELDS et même LDS, dans leur résolution de ce problème, passe à une troisième itération et ont un coût supérieur en nombre de nœuds développés, comme on peut l'observer sur les figures 3.5 et 3.6.

		X_1			X_2			X_3			X_4		
		1	2	3	4	5	6	7	8	9	10	11	12
X_1	1	○	○	○	○	●	○	○	○	○	○	○	○
	2	○	○	○	○	●	○	○	○	○	○	○	○
	3	○	○	○	●	○	○	○	○	○	○	○	○
X_2	4	○	○	●	○	○	○	○	●	○	○	○	●
	5	●	●	○	○	○	○	○	○	○	○	○	●
	6	○	○	○	○	○	○	○	●	○	○	○	●
X_3	7	○	○	○	○	○	○	○	○	○	●	●	○
	8	○	○	○	●	○	●	○	○	○	○	●	○
	9	○	○	○	○	○	○	○	○	○	●	●	○
X_4	10	○	○	○	○	○	○	●	○	●	○	○	○
	11	○	○	○	○	○	○	●	●	●	○	○	○
	12	○	○	○	●	●	●	○	○	○	○	○	○

○ instanciations compatibles

● instanciations incompatibles

Figure 3.7 – Matrice initiale du CSP

		X_1			X_2			X_3			X_4		
		1	2	3	4	5	6	7	8	9	10	11	12
X_1	1	○	○	○	● (rouge)	●	○	● (rouge)	○	○	○	○	○
	2	○	○	○	○	●	○	○	○	○	○	○	○
	3	○	○	○	●	○	○	○	○	○	○	○	○
X_2	4	● (rouge)	○	●	○	○	○	● (rouge)	●	○	○	○	●
	5	●	●	○	○	○	○	○	○	○	○	○	●
	6	○	○	○	○	○	○	○	●	○	○	○	●
X_3	7	● (rouge)	○	○	● (rouge)	○	○	○	○	○	●	●	○
	8	○	○	○	●	○	●	○	○	○	○	●	○
	9	○	○	○	○	○	○	○	○	○	●	●	○
X_4	10	○	○	○	○	○	○	●	○	●	○	○	○
	11	○	○	○	○	○	○	●	●	●	○	○	○
	12	○	○	○	●	●	●	○	○	○	○	○	○

○ instanciations compatibles

● instanciations incompatibles

● (rouge) instanciations compatibles avec probabilité

Figure 3.8 – Matrice du CSP après exploitation du no-good

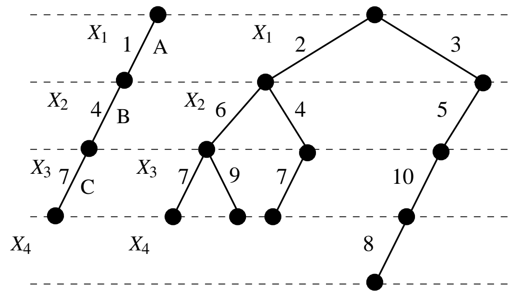


Figure 3.9 – Arbre de recherche développé par NG-YIELDS (15 nœuds)

3.5 Expérimentations

Dans cette partie, nous proposons de tester l'impact d'heuristiques exploitant les pondérations sur les variables et sur les valeurs proposées précédemment.

Les problèmes étudiés dans nos tests sont ceux de *car sequencing* issus de la [CSPLib] et des CSP aléatoires générés selon le modèle B de Frost *et al.* [Frost 1996]. Les critères d'évaluation sont le nombre de nœuds développés, noté *NEN* et le temps de calcul noté *CPU*. Tous les algorithmes ont été implémentés en C++. Ils ont été exécutés sous Linux Fedora sur un PC Core Duo 2.33 GHz ayant 3.37 Go de RAM.

3.5.1 Problèmes de *car sequencing*

Le problème de *car sequencing* est un problème d'ordonnancement particulier dont la résolution vise à déterminer l'ordre selon lequel des voitures doivent être réalisées sur une chaîne de fabrication [Hentenryck 1992b, Smith 1997]. Ce problème a fait l'objet d'une compétition de la ROADEF en 2005 [Boivin 2005, Solnon 2008]. Le problème de *car sequencing* se concentre sur une partie de la fabrication des véhicules, appelée montage, dans laquelle différents élements vont être ajoutés aux véhicules au sein de différentes unités de montage. Lors de cette étape, les voitures demandant des options particulières (*air bag*, toit ouvrant, etc.) vont nécessiter une plus grande charge de travail dans l'unité correspondante. Ces véhicules doivent donc être répartis dans la séquence des véhicules à fabriquer de telle sorte que la charge de travail ne dépasse pas la capacité des différentes unités de montage. Chaque voiture ne demande pas toutes les options

mais une liste d'options spécifiques. Chaque unité est caractérisée par une capacité limitée représentant le fait qu'elle peut produire au maximum r voitures avec l'option donnée sur une séquence de s voitures et nécessite un temps pour mettre au point l'option qui lui est associée.

Dans les expérimentations menées, nous avons considéré des instances de la CSPLib et plus particulièrement 70 instances satisfiables. Sur ces instances, nous allons comparer l'apport de différentes heuristiques d'ordre sur les variables et sur les valeurs au sein de la méthode backtrack chronologique (CB) et au sein d'une méthode de recherche à divergences limitées (LDS) avec application des divergences à partir du bas. Les propagations utilisées au sein de CB et de LDS sont du type forward-checking dans le sens où l'on propage la modification aux variables voisines et qu'on ne répercute pas ces modifications sur le reste du CSP. Nous avons en effet choisi de nous limiter à des propagations plus simples que celles proposées dans [Hentenryck 1992b] pour nous concentrer sur l'évaluation des méthodes et des heuristiques. Nous allons également étudier l'impact des modes de comptage des divergences (comptages binaire, non-binaire ou mixte) au sein d'une méthode LDS. On peut noter qu'en raison du fait que toutes les instances testées soient satisfiables, le mécanisme *DRestrict* proposé pour les méthodes à divergences ne peut être exploité.

Dans les tableaux suivants, *VarOrder* désigne l'heuristique d'ordre sur les variables, *ValOrder* l'heuristique d'ordre sur les valeurs utilisées, *#Solved* donne le nombre de problèmes résolus sur les 70 testés, *CPU* donne le temps de calcul moyen en secondes et *NEN* fournit la moyenne sur le nombre de nœuds développés. Nous donnons également les écarts-types du temps de calcul *stdev-CPU* et du nombre de nœuds développés *stdev-NEN*, ainsi que le nombre de problèmes pour lesquels le temps de calcul est inférieur au temps moyen *#Solved<CPU* et le nombre de problèmes pour lesquels le nombre de nœuds développés est inférieur au nombre de nœuds moyen *#Solved<NEN*.

La première partie des expérimentations compare l'impact des heuristiques sur les méthodes CB et LDS. Un temps limite d'exécution de 200 s a été fixé pour la résolution de chacun des problèmes. Les heuristiques d'ordre sur les variables utilisées sont `Lexico` et `Wvar`⊕`Lexico`. L'heuristique `Lexico` suit un ordre lexicographique.

L'heuristique `Wvar`$\oplus$`Lexico` choisit la variable X_i associée au plus grand poids $W_{var}(i)$ et, en cas d'égalité, considère l'ordre lexicographique.

Chacune de ces heuristiques d'ordre sur les variables a été associée à quatre heuristiques d'ordre sur les valeurs : `Lexico`, `MaxOpt`$\oplus$`Wval1`, `MaxOpt`$\oplus$`Wval2`, et `MaxOpt`$\oplus$`Wval3`. Ces trois dernières heuristiques de valeurs sélectionnent la valeur correspondant à la voiture qui nécessite le plus grand nombre d'options (`MaxOpt`) et, en cas de valeurs ex-aequo, choisissent la valeur ayant le plus grand poids calculé comme décrit dans le paragraphe 3.2.3.

Méthode	CB							
VarOrder	Lexico				Wvar⊕Lexico			
ValOrder :MaxOpt⊕	Lexico	Wval1	Wval2	Wval3	Lexico	Wval1	Wval2	Wval3
#Solved	36				37	36	36	36
CPU(s)	3.08				12.54	10.27	5.13	5.05
stdev-CPU	8.49				34.04	34.41	13.33	13.23
#Solved<CPU	30				34	31	30	30
NEN	700164				1995906	1888056	1030241	1030241
stdev-NEN	1974936				5734109	5884882	2759909	2759909
#Solved<NEN	30				34	31	30	30

Tableau 3.I – CB - Temps limite 200 s

Sur le tableau 3.I, nous observons qu'avec la méthode CB et l'heuristique de variables `Lexico`, il n'y aucune différence entre les quatre heuristiques de valeurs proposées. Nous pouvons expliquer cela par l'absence de valeurs ex-aequo après l'application de `MaxOpt`, chose qui fait qu'on n'utilise jamais les pondérations sur les valeurs, l'heuristique proposée pour les départager.

Si l'on considère les résultats obtenus par la méthode CB associée à l'heuristique de variables `Wvar`$\oplus$`Lexico`, on peut noter que l'influence des heuristiques de valeurs est relativement faible (le nombre de nœuds développés et le temps CPU sont plus faibles avec les heuristiques de valeurs utilisant les pondérations mais il y a un problème résolu de moins).

Pour les problèmes considérés, les heuristiques utilisant des pondérations de variables ou de valeurs proposées couplées à CB ne sont pas avérées intéressantes. Dans la suite, nous les testons intégrées dans une méthode à divergences.

La méthode à divergences proposée est évaluée avec un mode de comptage binaire

Methode	LDS							
VarOrder	Lexico				Wvar⊕Lexico			
ValOrder :MaxOpt⊕	Lexico	Wval1	Wval2	Wval3	Lexico	Wval1	Wval2	Wval3
#Solved	41				59			
CPU(s)	9.87	9.75	9.85	9.85	10.42	10.32	10.33	10.42
stdev-CPU	23.55	23.20	23.48	23.47	31.83	31.59	31.7	31.83
#Solved<CPU	35				50			
NEN	662301	360439			165075	165075		
stdev-NEN	2299353	960658			572800	572650		
#Solved<NEN	36	35	26	26	51			

Tableau 3.II – LDS - Comptage binaire - Temps limite 200 s

Méthode	LDS							
VarOrder	Lexico				Wvar⊕Lexico			
ValOrder :MaxOpt⊕	Lexico	Wval1	Wval2	Wval3	Lexico	Wval1	Wval2	Wval3
#Solved	38	42	47	46	65			
CPU(s)	11.81	17.21	15.06	19.56	9.61	9.44	9.46	9.64
stdev-CPU	36.98	42.04	28.47	35.53	34.66	33.77	33.85	34.80
#Solved<CPU	33	32	24	28	58			
NEN	541359	400542	82240	115377	314350			
stdev-NEN	1769914	256108	164659	217221	1396472			
#Solved<NEN	33	32	26	29	58			

Tableau 3.III – LDS - Comptage non-binaire - Temps limite 200 s

(tableau 3.II) et non-binaire (tableau 3.III) pour les divergences. Pour le comptage binaire, l'heuristique de variables Wvar⊕Lexico s'avère plus performante que l'heuristique Lexico : il y a un plus grand nombre de problèmes résolus, un nombre de nœuds développés plus faible pour un temps de calcul légèrement supérieur. En revanche l'apport des pondérations sur les valeurs n'est pas montré. En effet, les propagations étant responsables de l'absence de valeurs ex-aequo, les pondérations des valeurs ne sont pas exploitées. En ce qui concerne le comptage non-binaire, là encore l'heuristique de variables utilisant des pondérations est plus performante : plus de problèmes résolus dans un temps plus faible (même s'il y a plus de nœuds développés). Comme précédemment, l'utilisation de la pondération sur les valeurs ne montre pas son efficacité lorsqu'elle est couplée à une heuristique de variables utilsant la pondération. En revanche lorsque l'heuristique de variables est Lexico, les heuristiques de valeurs utilisant la pondération ont un impact sur les résultats : plus de problèmes résolus mais dans un temps de calcul plus élevé et un plus grand nombre de nœuds développés.

Que ce soit avec un mode de comptage binaire ou un mode de comptage non binaire,

la méthode LDS fournit de meilleurs résultats que la méthode CB.

Dans les tableaux 3.IV et 3.V, nous comparons la méthode LDS en mode de comptage binaire et non-binaire avec différentes limites pour la durée d'exécution : 200 s et 600 s. Pour ces comparaisons, nous avons retenu l'heuristique de variables avec pondération `Wvar`$\oplus$`Lexico` qui s'avère être la plus performante pour LDS et l'heuristique de valeurs `MaxOpt`$\oplus$`Lexico` car les heuristiques de valeurs basées sur les pondérations n'ont pas montré leur efficacité lorsqu'elles sont couplées à une heuristique de variables utilisant des pondérations. A la lecture de ces tableaux, on peut noter que le comptage non-binaire est toujours le meilleur pour ces problèmes de car sequencing avec plus de 95% de problèmes résolus pour une durée d'exécution maximale de 600 s.

Temps limite	#Solved	CPU(s)	stdev-CPU	#Solved<CPU	NEN	stdev-NEN	#Solved<NEN
200	59	10.42	31.83	50	165075	572650	51
600	62	26.75	86.48	54	376368	1149365	54

Tableau 3.IV – LDS - Comptage binaire

Temps limite	#Solved	CPU(s)	stdev-CPU	#Solved<CPU	NEN	stdev-NEN	#Solved<NEN
200	65	9.61	34.66	58	314350	1396472	58
600	67	21.76	65.3	62	941097	4231788	60

Tableau 3.V – LDS - Comptage non-binaire

A contrario de ce que certains auteurs ont pu déclarer sur la mauvaise performance des heuristiques dynamiques pour ce type de problèmes [Smith 1997], `Wvar`$\oplus$`Lexico` (heuristique dynamique) surclasse clairement les autres heuristiques testées.

Dans la seconde partie des expérimentations (tableaux 3.VI et 3.VII), nous testons l'intérêt d'utiliser un comptage mixte des divergences pour la méthode LDS. Nous utilisons un comptage non-binaire en haut (*LDS-NB-B*) ou en bas (*LDS-B-NB*) de l'arbre de recherche en faisant varier la profondeur de partitionnement de 1/5, 1/3 et 2/3 de la hauteur totale de l'arbre, mesurées à partir du haut de l'arbre. Le tableau 3.VI considère la méthode LDS avec l'heuristique d'ordre sur les variables `Lexico` et l'heuristique d'ordre sur les valeurs `MaxOpt`$\oplus$`Lexico`, et le tableau 3.VII considère la méthode LDS avec l'heuristique d'ordre sur les variable `Wvar`$\oplus$`Lexico` et l'heuristique d'ordre

sur les valeurs `MaxOpt`$\oplus$`Lexico`. Pour les instances étudiées, les résultats ne sont pas réguliers et n'améliorent pas les résultats obtenus avec le comptage non-binaire associé à LDS.

Methode(Temps limite)	LDS-NB-B(200)	LDS-NB-B(600)	LDS-B-NB(200)	LDS-B-NB(600)
Profondeur de partitionnement	#Solved			
1/5	49	61	39	45
1/3	62	64	40	44
2/3	46	47	41	46

Tableau 3.VI – LDS - VarOrder :`Lexico` - ValOrder :`MaxOpt`$\oplus$`Lexico` - Comptage mixte

Methode(Temps limite)	LDS-NB-B(200)	LDS-NB-B(600)	LDS-B-NB(200)	LDS-B-NB(600)
Profondeur de partitionnement	#Solved			
1/5	48	52	55	58
1/3	60	62	57	60
2/3	52	52	61	61

Tableau 3.VII – LDS - VarOrder :`Wvar`$\oplus$`Lexico` - ValOrder :`MaxOpt`$\oplus$`Lexico` - Comptage mixte

3.5.2 Problèmes aléatoires

Pour générer des CSP aléatoires, nous utilisons le générateur de modèle B [Frost 1996]. Les instances générées sont caractérisées par : n le nombre de variables, d la taille des domaines, la densité du problème, notée p_1 par la suite, correspondant au rapport du nombre de contraintes dans le graphe de contraintes et du nombre de toutes les contraintes envisageables, et la dureté du problème, notée p_2 par la suite, correspondant au rapport du nombre de n-uplets incompatibles et du nombre de tous les n-uplets envisageables. On fait varier la dureté du problème de manière à obtenir des instances autour du pic de complexité. Comme cela a été établi dans [Xu 2007], nous utilisons dans le générateur des densités qui permettent d'éviter les instances faciles dites *flowed*. En effet, autour du pic de complexité, les instances du modèle B sont beaucoup plus dures à résoudre que celles obtenues selon le modèle RB [Achlioptas 2001], ce qui en fait des instances *flowless* de celles générées à partir du modèle RB. Les instances de CSP aléatoires générées comportent soit 30 variables avec une taille de domaine fixe de

25 soit 40 variables avec une taille de domaine fixe de 20. La taille des échantillons est de 100 problèmes pour chaque quadruplet (n, d, p_1, p_2).

Le but des expérimentations est de comparer l'impact des heuristiques utilisant des pondérations au sein d'une méthode LDS et au sein de la méthode MAC. Les algorithmes de propagation employés dans MAC comme dans LDS sont des algorithmes d'arc-consistance de type AC-3 [Zhang 2001]. Les résultats obtenus sont évalués en termes de temps de calcul *CPU* exprimé en secondes et en termes de nombre de nœuds développés *NEN*.

Dans la première partie des expérimentations, nous comparons les résultats de la méthode MAC basée soit sur l'heuristique de variables `dom/Wvar` que nous avons proposée, soit sur l'heuristique de variables `dom/wdeg` (*cf.* chapitre 1). L'heuristique de valeurs utilisée est `min-conflict` (*cf.* chapitre 1). La figure 3.10 montre que les résultats obtenus par la méthode MAC avec l'heuristique de variables `dom/Wvar` sont meilleurs que ceux obtenus par MAC avec l'heuristique de variables `dom/wdeg`, à la fois en termes de temps de calcul qu'en termes de nombre de nœuds développés.

Nous évaluons ensuite l'impact des deux heuristiques de variables avec pondération `dom/Wvar` et `dom/wdeg` au sein d'une méthode de type LDS YIELDS présentée en 3.4.1 avec comptage binaire (figure 3.11) et avec comptage non binaire (figure 3.12). Comme précédemment, l'heuristique de valeur utilisée est `min-conflict`.

Dans le cas de la variante LDS avec comptage binaire (figure 3.11), l'heuristique `dom/Wvar` conduit à un plus faible nombre de nœuds développés. Le temps de calcul est également amélioré en moyenne avec cette heuristique de variables même si pour certaines instances LDS avec l'heuristique `dom/wdeg` est plus rapide.

En ce qui concerne la variante de LDS avec comptage non binaire (figure 3.12), l'utilisation de l'heuristique `dom/Wvar` s'avère plus efficace en termes de temps de calcul pour un nombre de nœuds développés relativement proche entre les deux heuristiques.

Nous évaluons également au sein des méthodes MAC, LDS avec comptage binaire, et LDS avec comptage non-binaire, les heuristiques `dom/Wvar` d'un côté (figure 3.13) ainsi que l'heuristique `dom/wdeg` d'un autre côté (figure 3.14). En utilisant l'heuristique `dom/Wvar` (figure 3.13), avec le comptage binaire, LDS n'améliore pas MAC. En

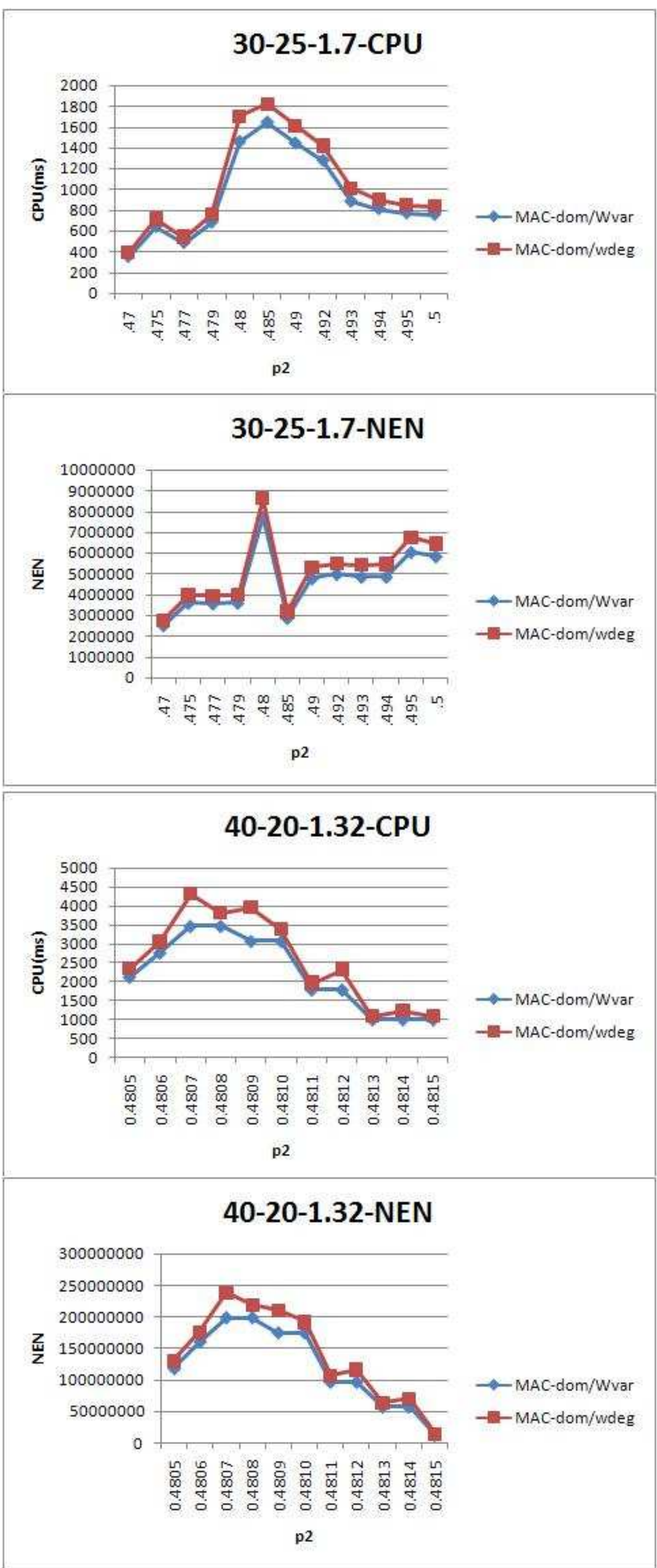


Figure 3.10 – Versions de MAC

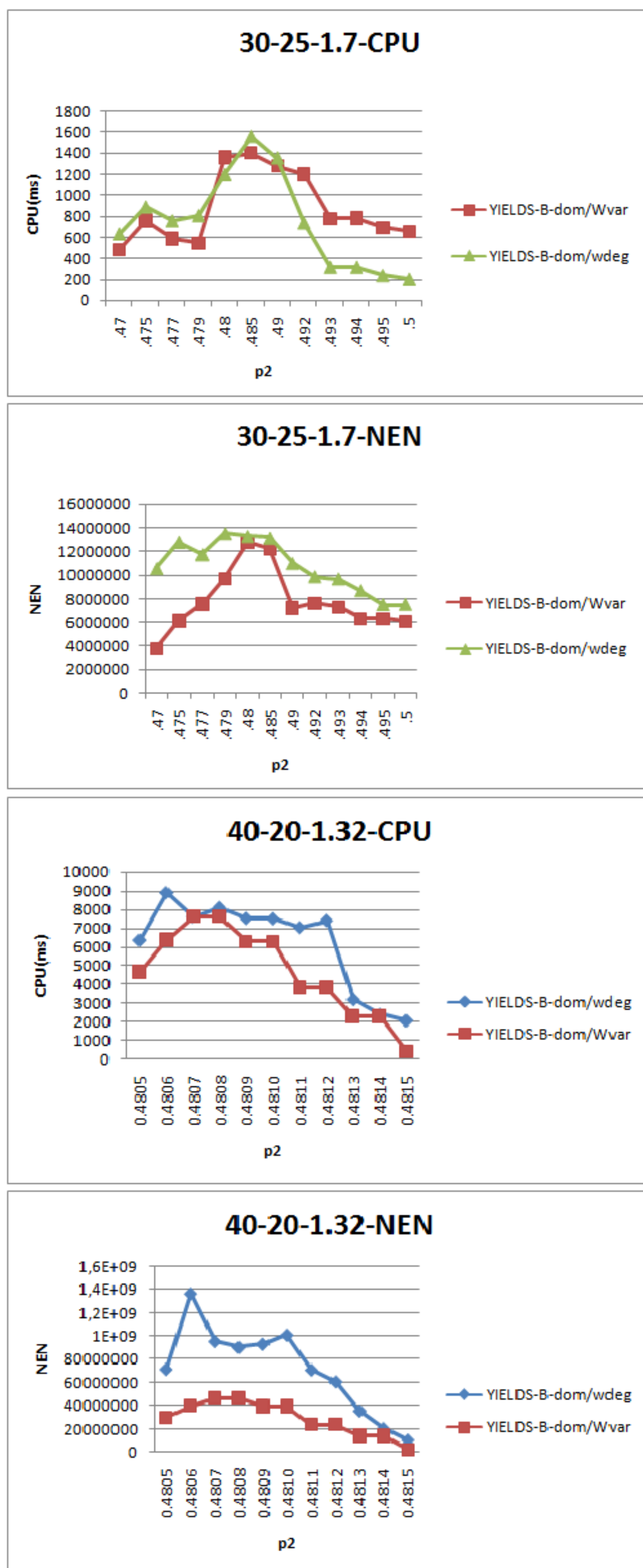


Figure 3.11 – Heuristiques de variables avec pondération avec comptage binaire

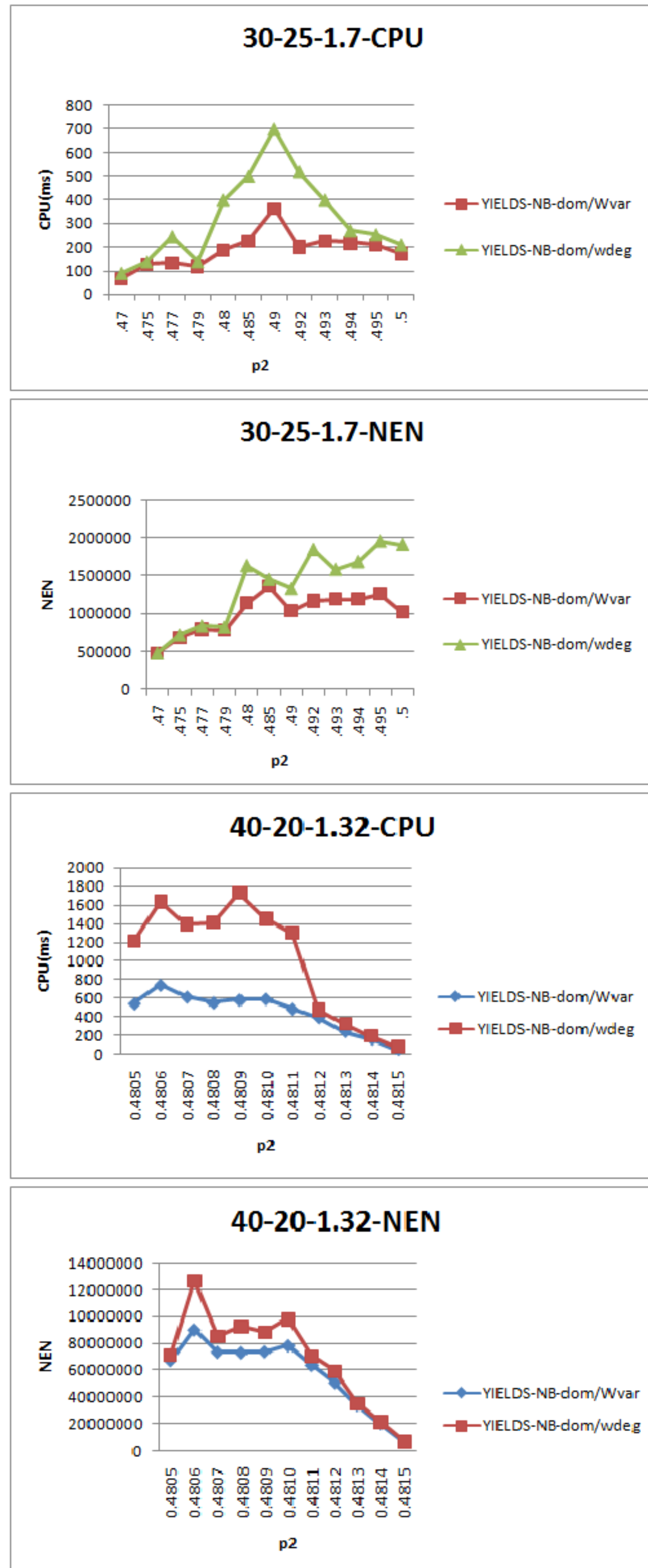


Figure 3.12 – Heuristiques de variables avec pondération avec comptage non-binaire

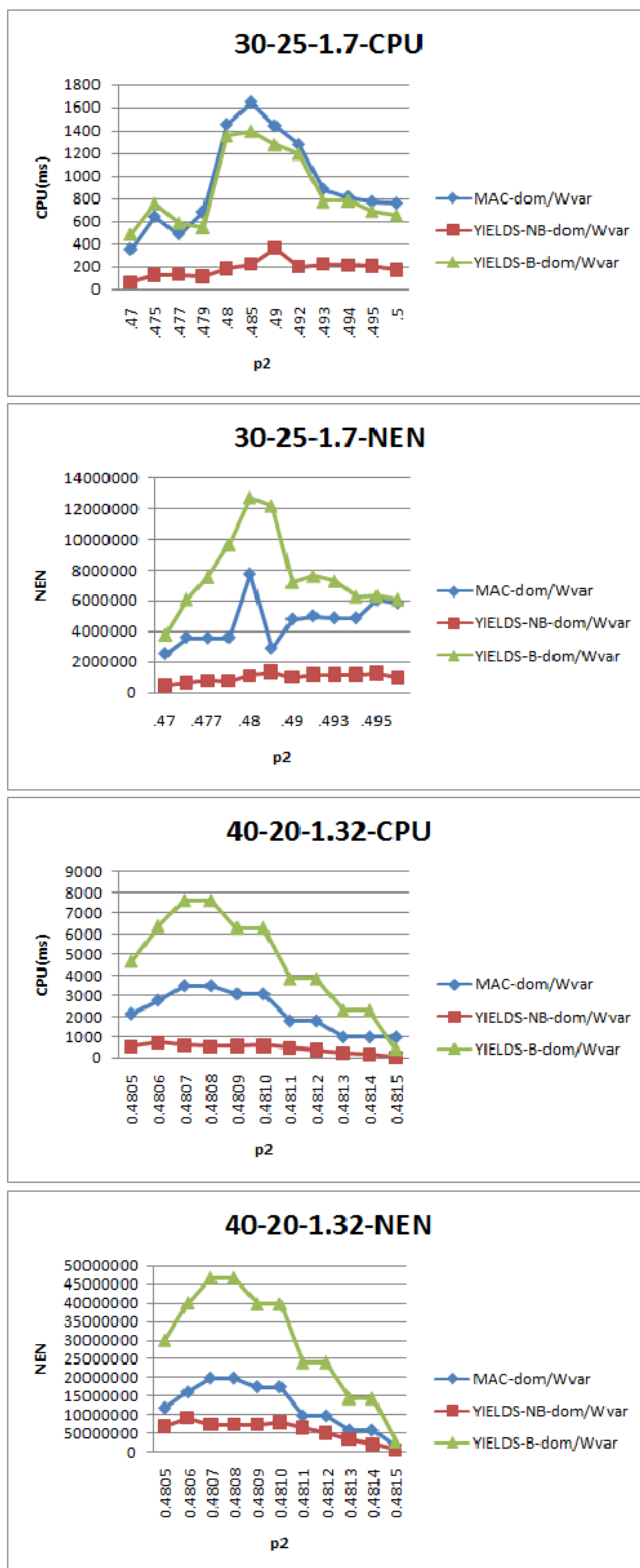


Figure 3.13 – Heuristique dom/Wvar dans différentes méthodes

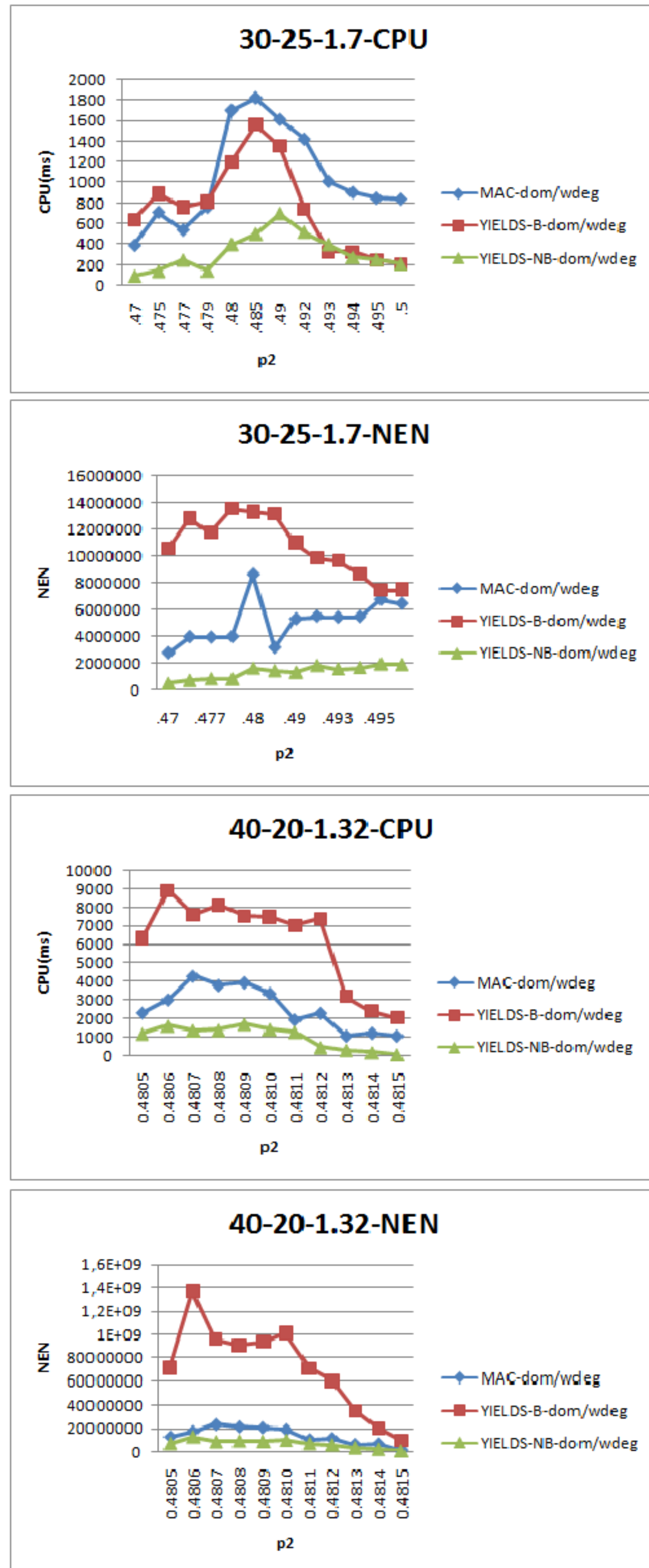


Figure 3.14 – Heuristique dom/wdeg dans différentes méthodes

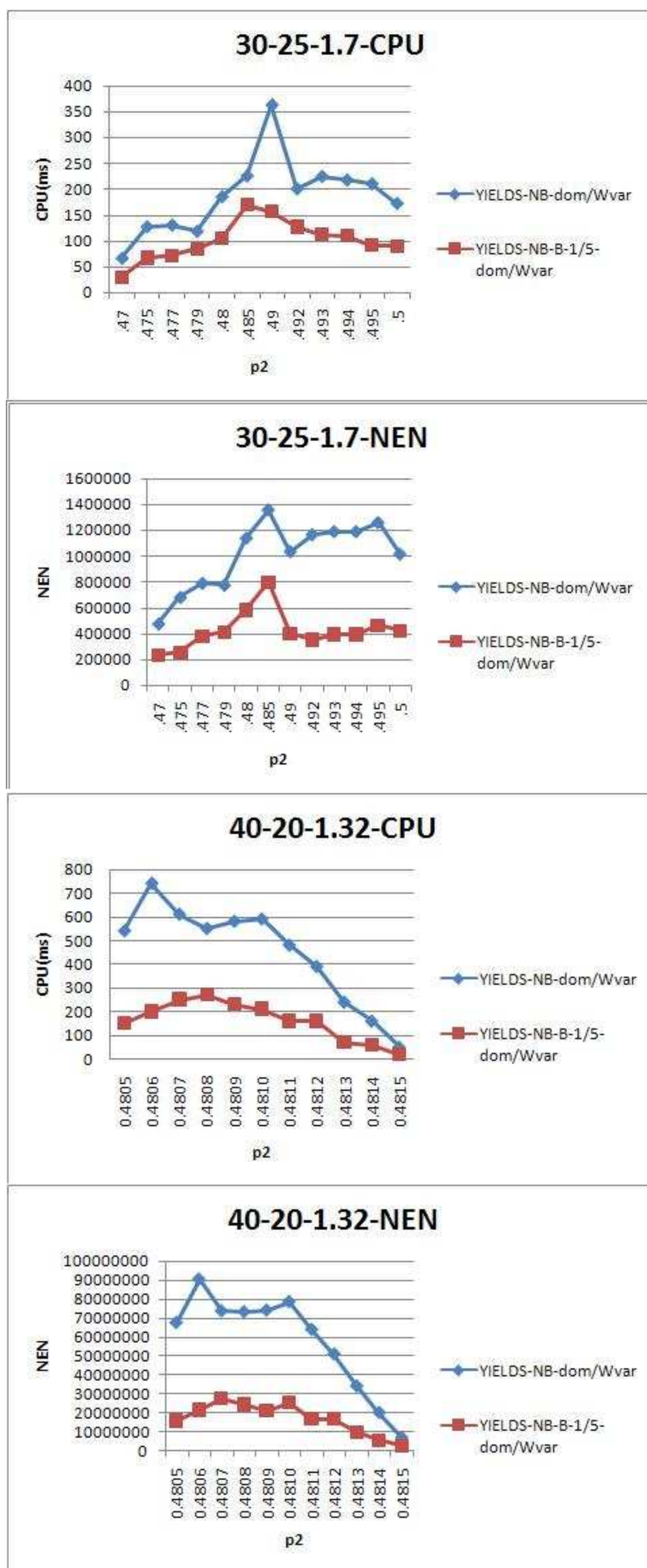


Figure 3.15 – Comptage mixte

revanche, LDS s'avère plus performante que MAC lorsqu'un comptage non-binaire est utilisé. En utilisant l'heuristique `dom/wdeg` (figure 3.14), avec le comptage non-binaire, LDS améliore nettement MAC et LDS avec comptage binaire. LDS avec le comptage binaire est globalement meilleure meilleure que MAC.

Dans ces expérimentations, les versions de LDS avec un comptage non-binaire sont meilleures que les autres versions en raison de l'efficacité d'une recherche de granularité fine sur de tels problèmes. L'efficacité de cette méthode s'explique par l'heuristique de pondération des variables considérée et par l'exploitation des restrictions sur les divergences. Ainsi, l'heuristique apprend des échecs rencontrés au cours de la résolution et le temps est réduit sur les problèmes insolubles. Au vu des résultats obtenus, nous nous concentrons dans la suite sur l'heuristique `dom/Wvar`.

La seconde partie des expérimentations concerne le mode de comptage mixte. Pour cela nous testons une variante de LDS avec le comptage non-binaire en haut et le comptage binaire en bas (LDS-NB-B) et une variante avec le comptage binaire en haut et non binaire en bas (LDS-B-NB). Nous faisons varier la profondeur de l'arbre à laquelle est effectué le changement de mode de comptage de 1/5, 1/3 et 2/3 (mesurées à partir du haut de l'arbre). Chaque variante de LDS pour chaque profondeur est comparée à LDS en comptage non-binaire.

Pour les instances étudiées, le mode de comptage mixte avec le comptage non-binaire en haut de l'arbre associé à une profondeur de 1/5 (*LDS-NB-B-1/5*) améliore les résultats comme nous pouvons l'observer sur la figure 3.15. Ce n'est pas le cas des autres profondeurs testées qui donnent des résultats irréguliers, moins intéressants que les résultats obtenus avec la version non-binaire de LDS. Cette piste est particulièrement prometteuse. En effet, quand la profondeur de partitionnement est modifiée, le temps CPU varie de plusieurs dizaines de secondes sur un unique problème, que ce soit en s'améliorant ou en se détériorant.

3.6 Conclusion

L'objectif majeur de ce chapitre a été de présenter quelques techniques qui peuvent contribuer à l'amélioration des méthodes de résolution de CSP de manière générale ou à l'amélioration des méthodes basées sur les divergences. La plupart de ces techniques ont été évaluées dans la seconde partie de ce chapitre. Les résultats expérimentaux obtenus attestent de l'efficacité de certaines de ces techniques notamment de l'utilisation de la pondération de variables au sein d'heuristiques d'instanciation mais aussi du mode de comptage des divergences.

Les expérimentations permettent notament d'évaluer les modes de comptage des divergences et l'efficacité des propositions que nous avons faites concernant les heuristiques d'instanciation se basant sur les poids (`Wvar`, `Wval`). Concernant les problèmes de car sequencing, nous concluons que la méthode LDS avec `Wvar`$\oplus$`Lexico` comme heuristique d'ordre sur les variables obtient les meilleurs résultats. Concernant les problèmes aléatoires, nous constatons que MAC associé à `dom/Wvar` donne globalement de meilleurs résultats que ceux obtenus par MAC avec `dom/wdeg`. En comparant différentes heuristiques (`dom/wdeg` et `dom/Wvar`) dans une méthode de type LDS (YIELDS), considérés en comptage binaire et non-binaire, nous obtenons que l'heuristique `dom/Wvar` donne des résultats meilleurs surtout en comptage non-binaire.

Ce chapitre s'est focalisé sur des méthodes complètes dédiées à la satisfaction de contraintes. Dans le chapitre suivant, nous nous penchons sur des méthodes visant la résolution de problèmes d'optimisation.

CHAPITRE 4

ADAPTATION DES MÉTHODES À DIVERGENCES AUX PROBLÈMES D'OPTIMISATION : CAS DES PROBLÈMES D'ORDONNANCEMENT AVEC TIME-LAGS

Dans ce chapitre nous abordons la résolution de problèmes d'optimisation par les méthodes à divergences. Nous nous intéressons plus particulièrement à la méthode *Climbing Discrepancy Search*. Différentes variantes de cette méthode sont étudiées pour la résolution des problèmes d'ordonnancement avec time-lags. Des expérimentations menées sur des instances de la littérature attestent de l'efficacité de certaines de nos propositions.

Certains des résultats proposés dans ce chapitre ont été présentés dans les conférences internationales ISCO'10 [Karoui 2010a], PMS'10 [Karoui 2010c], ainsi que MOSIM'10 [Karoui 2010d] et ROADEF'10 [Karoui 2010b].

4.1 Méthodes à base de divergences pour l'optimisation

Plusieurs travaux ont adopté le principe de recherche LDS [Harvey 1995b] pour traiter des problèmes d'optimisation. Parmi ces travaux, nous pouvons citer ceux de [Levasseur 2007] qui s'intéressent à la résolution de problèmes de satisfaction de contraintes valués (*Weighted Constraint Satisfaction Problems* ou WCSP), ainsi que ceux de [Hmida 2007] et de [Gacias 2008] qui s'intéressent à la résolution de problèmes d'ordonnancement. Ces travaux exploitent le parcours spécifique de l'espace de recherche que permet une méthode à divergences et obtiennent des résultats intéressants. Dans la suite, nous détaillons quelques-uns de ces travaux.

4.1.1 Climbing Discrepancy Search (CDS)

La méthode CDS [Milano 2002], ou *recherche par montée de divergences*, est une adaptation du principe de recherche de la méthode LDS à un contexte d'optimisation. C'est une méthode d'optimisation approchée dans le sens où elle chercher à déterminer une solution de bonne qualité par rapport à une fonction objectif donnée. La méthode CDS détermine initialement une première solution à l'aide d'une heuristique. Cette solution est évaluée à l'aide de la fonction objectif considérée. Puis, lors de l'itération suivante, la méthode CDS détermine les solutions présentant une divergence par rapport à la solution de référence. Si ces solutions trouvées n'améliorent pas le coût de la solution de référence, CDS augmente le nombre de divergences à considérer. En revanche dès qu'une solution améliore la valeur de la fonction objectif de la solution de référence, la méthode CDS considère que la nouvelle solution trouvée devient la nouvelle solution de référence et le compteur de divergences est remis à zéro. L'algorithme 27 illustre ce principe. Ainsi, la méthode CDS peut être vue comme une méthode explorant un voisinage d'une solution de plus en plus grand afin d'améliorer la qualité d'une solution de référence. On peut donc l'apparenter à une méthode de recherche à voisinage variable telle que présentée dans [Hansen 2001]. Par rapport à une méthode à voisinage variable, la spécificité de CDS réside dans la définition du voisinage qui repose sur la notion de divergence.

Algorithme 27 : CDS(X, D, C, f, Sol)

1 $k \leftarrow 0$ % k est le nombre de divergences
2 $k_{max} \leftarrow n$ % n est le nombre de variables
3 $Sol \leftarrow Solution_Initiale(X,D,C)$ % Sol est la solution initiale
4 **tant que** $k \leq k_{max}$ **faire**
5 $\quad k \leftarrow k+1$
6 $\quad F \leftarrow Generer_Feuilles(Sol,k)$ %Générer l'ensemble F des feuilles à k divergences par rapport à Sol
7 $\quad Sol' \leftarrow Meilleure_Feuille(F)$
8 $\quad$ **si** $Est_Meilleur(f(Sol'),f(Sol))$ **alors**
9 $\quad\quad$ % Mise à jour de la solution initiale
10 $\quad\quad Sol \leftarrow Sol'$
11 $\quad\quad k \leftarrow 0$

4.1.2 Exemple illustratif

Soit un problème de minimisation comportant trois variables de décision, chacune d'elles pouvant prendre deux valeurs possibles. La figure 4.1 illustre le parcours suivi par CDS pour traiter ce problème et donne l'ordre de génération des feuilles de l'arborescence. L'instanciation initiale correspond à une solution S_{init1} ayant une valeur de la fonction objectif notée $f(S_{init1})$. La première itération est associée à la recherche des solutions ayant une divergence : nous supposons sur cet exemple que nous ne rencontrons pas de solution de meilleure qualité que $f(S_{init1})$. A la deuxième itération associée à deux divergences, nous supposons que nous rencontrons une solution S meilleure ($f(S) < f(S_{init1})$). Cette solution S est adoptée comme nouvelle solution de référence (S_{init2}) et la méthode CDS recommence la recherche en remettant le nombre de divergences à zéro. Lors de la prochaine itération à 1 divergence, CDS trouve de nouveau une solution de meilleure qualité. Les itérations se poursuivent jusqu'à ce qu'on parcourt la totalité de l'espace de recherche autour d'une solution de référence ou, plus concrètement, qu'une limite sur le temps ou le nombre de divergences soit atteinte. On peut également abandonner la recherche si l'on ne trouve plus d'amélioration pendant un certain temps ou un nombre d'itérations convenus au préalable.

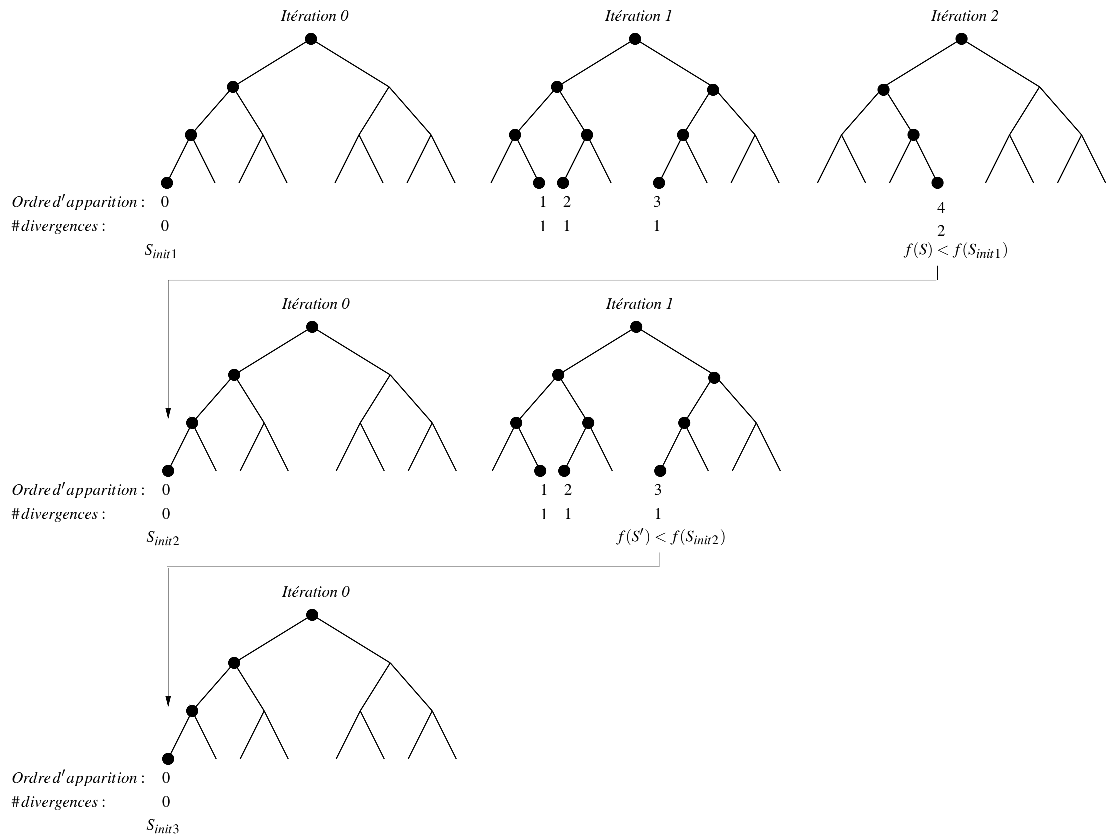


Figure 4.1 – Arbres de recherche développés par CDS

4.1.3 Climbing Depth-bounded Discrepancy Search (CDDS)

La méthode CDDS, ou *recherche par montée de divergences limitée par la profondeur*, proposée par [Hmida 2007] se base à la fois sur la méthode CDS pour explorer le voisinage d'une solution de référence et sur la méthode DDS afin de limiter l'exploration des voisinages par la profondeur des divergences. Pour cela, lors du calcul de la procédure $Generer_Feuilles(Sol,k)$ de l'algorithme 27 seules les solutions à k divergences situées au plus à une profondeur d sont déterminées. Ceci peut être envisagé de différentes manières :

- la profondeur d est telle que $1 \leq d \leq k$. Dans ce cas, les solutions à k divergences sont au plus à la profondeur k dans l'arbre de recherche. Ainsi, la recherche descend en prodondeur dans l'arbre au fur est à mesure que la valeur k du nombre maximum de divergences augmente ;

– la profondeur d est limitée a priori par une valeur donnée (dépendant éventuellement des caractéristiques du problème à résoudre). Dans ce cas, les solutions à k divergences sont toutes au plus à la profondeur d fixée dans l'arbre de recherche et seule la partie de l'arborescence située au dessus de cette profondeur d est explorée durant la résolution.

4.1.4 Autres adaptations de la méthode CDS

Plusieurs mécanismes peuvent été greffés à ce principe de recherche. Dans [Hmida 2007], des calculs de bornes inférieures de la fonction objectif (minimisation) sont réalisés afin d'élaguer des parties de l'espace de recherche. L'efficacité de la méthode CDDS incluant des calculs de bornes a été attestée par son application à des problèmes de flowshop hybride généraux [Hmida 2007] ou à des problèmes de flowshop hybride à deux étages pour lesquels les résultats obtenus font partie des meilleurs résultas de la littérature.

Dans [Hmida 2010, Huguet 2004], des mécanismes permettant de limiter les points d'application des divergences sont aussi proposés. Ainsi certaines variables et/ou certaines valeurs sont considérées de manière prioritaire pour effectuer des divergences. Ces mécanismes généraux ont été validés par des expérimentations sur des problèmes de jobshop flexible sur lesquels ils se sont avérés très performants.

Dans [Gacias 2008], une méthode mixte entre CDS et CDDS est proposée, appelée HD-CDDS. Jusqu'à un certain nombre de divergences, la méthode CDS est appliquée. Une fois que ce nombre limite de divergences est atteint, seuls certains niveaux de l'arbre sont explorés avec un nombre croissant de divergences. Cette méthode a montré son efficacité pour la résolution de problèmes d'ordonnancement à machines parallèles avec contraintes de précédence et temps de préparation en considérant la minimisation de la durée totale de l'ordonnancement et celle de la somme des dates de fin des jobs en améliorant les résultats déjà fournis par une méthode de type LDS [Néron 2008].

Toujours dans [Gacias 2008], des règles de dominance ont été introduites avec succès au sein d'une méthode à divergence afin d'élaguer certaines branches de l'arborescence.

4.2 Problèmes d'ordonnancement avec contraintes de délais

4.2.1 Définitions

Nous nous intéressons dans cette partie à la résolution de problèmes d'ordonnancement de type flowshop et jobshop avec contraintes de délais plus connues sous le nom de *time lags*. Les problèmes d'ordonnancement étudiés sont composés d'un ensemble de n travaux (jobs) devant être réalisés sur un ensemble de m ressources (machines). Chaque job i comporte n_i opérations (ou tâches) devant être exécutées en séquence. Une opération est réalisée sans interruption par une seule ressource. Les ressources sont disjonctives et ne peuvent donc exécuter qu'une seule opération à la fois. L'objectif considéré est la minimisation de la date de fin de l'ensemble des jobs (makespan $C_{\max}$). Avant d'introduire la notion de time lags, nous donnons les spécificités des problèmes de flowshop et de jobshop.

Flowshop Dans un problème de flowshop, tous les jobs passent sur les machines dans le même ordre, *i.e.,* la gamme de réalisation des opérations des jobs est unique. La différence éventuelle des durées de passage sur chaque machine constitue l'unique point qui peut distinguer deux jobs différents.

Jobshop Dans un problème de jobshop, l'ordre de passage sur les machines peut différer d'un job à un autre (gammes multiples). On peut même envisager qu'un job passe plusieurs fois par une même machine (recirculation) ou bien qu'il n'y passe pas du tout. Le flowshop est donc un cas particulier de jobshop.

Time lags Les time lags sont des contraintes temporelles spécifiques qui peuvent apparaître dans un problème d'ordonnancement. Ils ont été introduits pour la première fois par Mitten en 1958 [Mitten 1958]. [Dell'amico 1996] les définit comme étant le temps entre la fin d'une opération et le début d'une autre.

Dans notre cas, il s'agit de contraintes de délais uniquement entre deux opérations consécutives d'un même job. Ainsi, la distance temporelle séparant ces opérations comporte une limite inférieure et une limite supérieure qu'on appelle respectivement contrainte de time lag minimum et contrainte de time lag maximum. Ces contraintes viennent se rajouter aux contraintes initiales du problème considéré.

4.2.2 Modélisation

Une formulation possible du problème de jobshop ou de flowshop avec time lags est la suivante.
Soient $i = 1,...,n$ l'indice des jobs, $j = 1,...,n_i$ l'indice des opérations du job i, (i,j) désigne la $j^{ème}$ opération du job j, $m_{i,j}$ correspond à la machine allouée à la tâche (i,j), $p_{i,j}$ est la durée de l'opération (i,j), $t_{i,j}$ représente la date de début de l'opération (i,j) qu'il faut déterminer, $TLmin_{i,j,j+1}$ et $TLmax_{i,j,j+1}$ désignent respectivement les valeurs des time lags minimum et maximum entre les opérations (i,j) et $(i,j+1)$.
Les contraintes du problème s'écrivent alors comme suit :

$$C_{max} \geq t_{i,j} + p_{i,j} \quad \forall i = 1,\ldots,n, j = 1,\ldots,n_i, \tag{4.1}$$

$$t_{i,j+1} \geq t_{i,j} + p_{i,j} + TLmin_{i,j,j+1} \quad \forall i = 1,\ldots,n, j = 1,\ldots,n_{i-1}, \tag{4.2}$$

$$t_{i,j+1} \leq t_{i,j} + p_{i,j} + TLmax_{i,j,j+1} \quad \forall i = 1,\ldots,n, j = 1,\ldots,n_{i-1}, \tag{4.3}$$

$$t_{i,j} \geq t_{k,l} + p_{k,l} \text{ ou } t_{k,l} \geq t_{i,j} + p_{i,j} \quad \forall (i,j),(k,l) \text{ t.q. } m_{i,j} = m_{k,l}, \tag{4.4}$$

$$t_{i,j} \geq 0 \quad \forall i = 1,\ldots,n, j = 1,\ldots,n_i. \tag{4.5}$$

La première contrainte permet de déterminer les dates de fin des opérations. Les deuxième et troisième contraintes représentent les contraintes de durée et de time lags minimum et maximum. La quatrième contrainte traduit le partage de ressource : deux opérations nécessitant la même machine ne peuvent s'exécuter en même temps. La dernière contrainte est la contrainte sur la date de début minimale des différentes opérations.

De manière globale, les problèmes intégrant des contraintes de time lags sont de difficulté supérieure ou égale à celle du problème initial considéré sans time lags, sachant que les problèmes de jobshop et de flowshop sont NP-difficiles dans le cas général.

Une difficulté spécifique de ces problèmes provient de la contrainte de time lags maximum qui peut créer une impossibilité de déplacer une opération plus tard sur une ressource, ce qui resserre les créneaux disponibles pour l'insertion des opérations. Les problèmes contenant uniquement des contraintes liées aux time lags minimum sont moins problématiques car de telles contraintes peuvent être absorbées en rallongeant

la durée de l'opération précédente.

4.2.3 Méthodes de résolution de la littérature

Peu de méthodes de résolution ont été utilisées pour résoudre les problèmes de jobshop ou de flowshop avec time lags. Nous pouvons citer une méthode tabou [Caumond 2004], un algorithme mémétique [Caumond 2008], ainsi qu'un Branch and Bound intégrant des techniques de propagation de contraintes [Huguet 2010].

Dans [Caumond 2004], une adaptation de la méthode tabou proposée par Nowicki et Smutnicki [Nowicki 1996] pour le jobshop a été présentée pour la résolution de problèmes de jobshop avec time lags. L'adaptation proposée concerne principalement la mise en place d'une heuristique intégrant les contraintes de time lags.

Dans [Caumond 2008], un algorithme mémétique adapté aux problèmes de jobshop et de flowshop avec time lags est proposé. Cette méthode se révèle efficace pour les flowshop et pour les jobshop avec time lags serrés et notamment au cas de problèmes sans attente. Dans la partie expérimentations, nous nous comparons aux résultats obtenus par cet algorithme.

Dans [Huguet 2010], un Branch and Bound intégrant des techniques de propagation de contraintes spécifiques a été proposé ainsi qu'une nouvelle heuristique pour la résolution des problèmes de jobshop avec time lags. Les résultats obtenus par le Branch and Bound se limite cependant à des instances de petite taille.

4.3 Méthode à base de divergences pour la résolution des problèmes d'ordonnancement avec time-lags

4.3.1 Adaptation de la méthode CDS

Nous proposons dans cette partie d'utiliser la méthode CDS présentée précédemment pour les problèmes d'ordonnancement avec time lags considéré. Nous étudions plusieurs paramétrages de cette méthode. Tout d'abord, les divergences peuvent être effectuées en haut de l'arborescence d'abord ou en bas d'abord. Nous proposons également d'introduire une pondération des variables considérées par l'heuristique d'instanciation afin

d'intégrer un mécanisme d'apprentissage des "échecs" rencontrés lors des différentes itérations de la méthode.

Par ailleurs, pour les expérimentations, la méthode CDS sera limitée par un temps CPU alloué.

4.3.2 Positionnement des divergences

Plusieurs variantes sont possibles pour fixer l'emplacement dans l'arbre de recherche des divergences que nous souhaitons appliquer. Une première variante possible consiste à diverger en priorité en haut de l'arbre ou, alternativement, en bas de l'arbre. Nous avons également testé de diverger uniquement dans une partie bien déterminée de l'arbre de recherche, par exemple en haut de l'arborescence, et de continuer la recherche dans les autres parties de l'arbre sans procéder à davantage de divergences.

4.3.3 Heuristique d'initialisation utilisée

L'heuristique utilisée pour l'obtention d'une solution admissible (*i.e.,* respectant les contraintes) est celle proposée dans [Huguet 2010]. Elle consiste à sélectionner progressivement les jobs et à placer au plus tôt l'ensemble des opérations du job sélectionné sur les machines en respectant les contraintes de disponibilité des ressources et de séquencement entre ces opérations. En cas d'infaisabilité pour le placement d'une opération sur une machine, celle-ci est décalée dans le temps. Ce décalage d'une opération donnée peut entraîner le décalage de l'opération précédente afin de respecter les contraintes de time lags. Dans le pire des cas, les différents jobs sont ordonnancés les uns à la suite des autres. Cette heuristique s'appuie sur différents classements des jobs (en fonction de leur durée, de leur time lags ou d'une composition entre les deux) qui sont statiques.

Lors du déroulement de la méthode CDS avec cette heuristique, les variables à instancier représentent la sélection des jobs. La sélection d'un job s'effectue en fonction d'un classement sur les jobs (durée, time lags, ...) et une divergence consiste alors à sélectionner un autre job.

Cette heuristique est appliquée dans l'exemple de la figure 4.2 qui considère un

jobshop comportant trois jobs. Le classement des jobs dans cet exemple suit un ordre lexicographique.

4.3.4 Pondération des variables

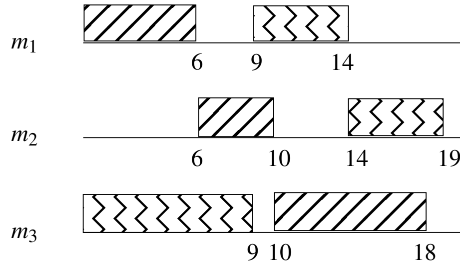
Dans le chapitre 3, nous avons introduit un mécanisme de pondération de variables au sein d'une méthode à divergences dans un contexte de satisfaction de contraintes. Suivant le principe "first fail", ces pondérations permettent de guider la recherche vers des parties difficiles du problème pour accélérer sa résolution. Dans cette partie, nous introduisons un mécanisme de pondération similaire pour l'optimisation.

Dans un contexte de satisfaction de contraintes, les pondérations de variables consistent à augmenter le poids d'une variable ayant rencontré un échec. Pour la résolution de problèmes d'optimisation, les méthodes arborescentes ne rencontrent pas d'échec (toute feuille est une solution). Néanmoins, lors de l'application d'une heuristique d'instanciation, il est possible de collecter des informations sur les variables du problème afin d'introduire des pondérations sur les variables.

Pour les problèmes d'ordonnancement avec time lags considérés, compte tenu de l'heuristique d'instanciation utilisée (basée sur l'insertion progressive de jobs dans un ordonnancement partiel) nous pouvons associer des poids aux différents jobs. Ainsi pour un job J_i, nous associons un poids W_i ; initialement les valeurs des poids sont identiques pour tous les jobs. Lors du placement des opérations du job J_i sélectionné par l'heuristique, certaines opérations peuvent être décalées dans le temps compte tenu des contraintes de time lags. Ces décalages dans le temps des différentes opérations d'un job peuvent être interprétées comme des "échecs" par rapport au fait de pouvoir positionner ces opérations au plus tôt et vont entraîner des incrémentations du poids du job correspondant. Plusieurs modes d'incrémentation des poids d'un job, peuvent être définis :

– **Mode A)** Chaque décalage d'une opération d'un job provoque l'incrémentation du poids du job. Dans l'exemple de la figure 4.2, lors du placement de l'opération $(3,2)$ une infaisabilité est détectée par rapport à la contrainte de time lags maximum avec l'opération $(3,1)$. Les opérations $(3,2)$ et $(3,1)$ sont alors décalées dans le temps. Lors du placement de l'opération $(3,3)$, une infaisabilité apparaît en rai-

J_1 et J_2 déjà ordonnancés :



$J_1 = \{(O_{11}, m_1, 6), (O_{12}, m_2, 4), (O_{13}, m_3, 8)\}$

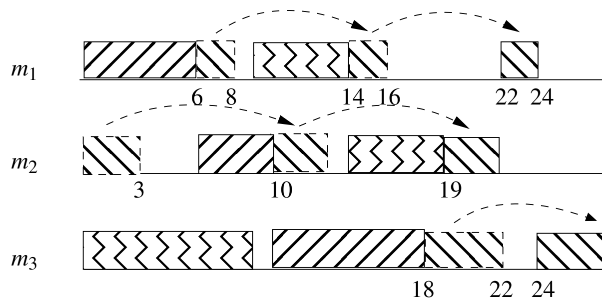
$J_2 = \{(O_{21}, m_3, 9), (O_{22}, m_1, 5), (O_{23}, m_2, 5)\}$

$J_3 = \{(O_{31}, m_2, 3), (O_{32}, m_1, 2), (O_{33}, m_3, 4)\}$

O_{31}-O_{32} : $TLmin$=0 and $TLmax$ = 1

O_{32}-O_{33} : $TLmin$=0 and $TLmax$ = 0

Insertion de J_3 :



		A	B	C
# d'incrémentations correspondant aux décalages de	O_{31}	+2	+1	true
	O_{32}	+2	+1	true
	O_{33}	+1	+1	true
Poids associé au job	J_3	+5	+3	+1

Figure 4.2 – Différentes possibilités d'incrémentation du poids associé à un job

son du time lag maximum entre $(3,3)$ et $(3,2)$, ces deux opérations sont alors décalées. Le décalage de l'opération $(3,2)$ entraîne alors un nouveau décalage de l'opération $(3,1)$ pour respecter les contraintes de time lags. Ainsi, l'opération $(3,1)$ est décalée deux fois sur la machine m_2, l'opération $(3,2)$ est décalée deux fois sur la machine m_1 et l'opération $(3,3)$ est décalée une fois. Le poids du job J_3 est alors incrémenté de $2+2+1=5$. Ce mode d'incrémentation est qualifié d'*incrémentation par opération*.

- **Mode B)** Le poids d'un job est incrémenté à chaque fois qu'une de ses opérations n'est pas insérée au plus tôt sur la machine qu'elle nécessite. Avec ce mode d'incrémentation des poids, il y a au plus une incrémentation par opération (ou par machine si toutes les opérations demandent des machines distinctes). Sur l'exemple de la figure 4.2, les opérations $(3,1)$, $(3,2)$ et $(3,3)$ n'ont pu être placées dans le premier intervalle libre sur les différentes machines, le poids du job J_3 est alors incrémenté de 3. Ce mode d'incrémentation est qualifié d'*incrémentation par machine*.
- **Mode C)** Le poids d'un job est incrémenté dès qu'il y a au moins un décalage sur une de ses opérations. Ce mode d'incrémentation est un mode binaire : soit il y a au moins un décalage et le poids est incrémenté de 1 soit il n'y a aucun décalage des différentes opérations et le poids du job n'est pas incrémenté. Sur l'exemple de la figure 4.2, avec ce mode d'incrémentation, le poids du job J_3 est augmenté de 1 car il y a eu au mois un décalage pour un de ses opérations. Ce mode d'incrémentation est qualifié d'*incrémentation par job*.

4.3.5 Exploitation des pondérations des variables

Lors d'une itération de la méthode CDS, la méthode recherche les solutions voisines ayant k divergences par rapport à la solution courante. Plusieurs feuilles vont être générées et lors du calcul de chaque feuille différentes pondérations vont être associées aux jobs.

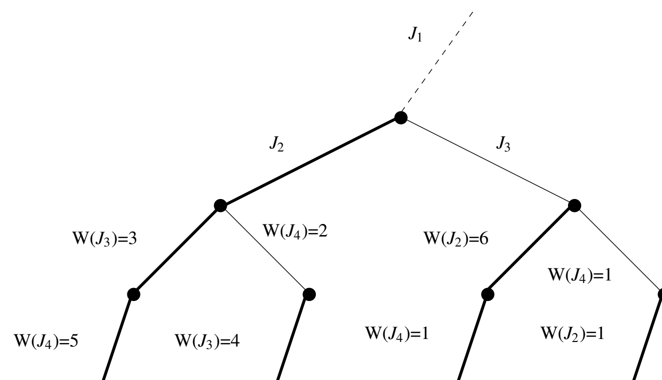
Avant de passer à la prochaine itération (recherche des solutions à $k+1$ divergences si aucune solution à k divergences n'améliore la solution courante ou dans le cas contraire

recherche d'une solution à 0 divergence par rapport à une nouvelle solution courante), les différentes pondérations sur les jobs doivent être combinées entre elles.

On peut alors envisager deux combinaisons possibles :

- **Somme** Le poids d'un job pour la nouvelle itération est la somme des pondérations de ce job sur toute l'itération.
- **Max** Dans ce cas le poids d'un job pour la nouvelle itération correspond au maximum des poids obtenus par ce job lors de l'itération.

La figure 4.3 représente une itération donnée de la méthode CDS. Lors de cette itération, les jobs J_2, J_3 et J_4 ont obtenu différentes pondérations dans chacune des branches. Avant de passer à la nouvelle itération, le poids de chacun des jobs est déterminé en appliquant une des combinaisons, *Somme* ou *Max*.



	J_2	J_3	J_4
Somme	7	7	9
Max	6	4	5

Figure 4.3 – Modes de comptage des poids sur les jobs

Dans les itérations suivantes, les poids des jobs sont considérés de manière complémentaire à l'ordre utilisé pour classer les jobs (durées décroissantes, time lags décroissants, ...). Par exemple, on peut définir une nouvelle heuristique réalisant un classement des jobs dans l'ordre décroissant des produits durée$\times$poids.

4.4 Expérimentations

4.4.1 Benchmarks

Pour tester les différents paramètres proposés pour une méthode CDS, nous avons considéré les instances de flowshop et de jobshop avec time lags proposés par [Caumond 2008]. Ces instances correspondent à des problèmes classiques de flowshop (instances de Carlier : $\{carY\}_{Y=5..8}$) et de jobshop (instances $\{laX\}_{X=1..20}$ de Lawrence et instances $ft06$ et $ft10$ de Fisher et Thompson) modifiées pour y introduire des contraintes de time lags. Seules les contraintes de time lags maximum ont été introduits par le biais d'un coefficient $\alpha \in \{0; 0.25; 0.5; 1; 2; 3; 5; 10\}$. A l'aide de ces coefficients, les valeurs des time lags maximum des opérations d'un job j donné sont déterminés de la manière suivante : $TLmax_{i,j,j+1} \leftarrow \alpha \times (\sum_{j=1}^{n_i} p_{i,j})/n_i$. Ainsi, dans les instances considérées, pour un job j, les contraintes de time lags sont identiques entre toutes les opérations qui le composent ; de plus la valeur de time lags maximum est liée à la durée du job.

La méthode CDS a été développée en Ada 95 avec le compilateur GNAT 4.4.1 et les tests ont été effectué sur un PC à 2.33 GHz avec 4 GB de RAM fonctionnant sous Linux Red Hat 4.4.1-2.

La méthode CDS a été limitée par un temps de calcul maximum de 200 secondes.

4.4.2 Résultats obtenus

Comparaison des heuristiques de génération de la solution initiale

Nous avons testé différents ordres de classement des jobs pour l'heuristique de génération de la solution initiale. Ces heuristiques se basent sur les durées des jobs (*D*), sur les durées des time lags (*DTL*), sur la somme des durées des jobs et des time lags (*D+DTL*) ainsi que leur rapport (*D/DTL*), et l'ordre lexicographique.

Les résultats des tests montrent que l'heuristique de classement des jobs basée sur l'ordre décroissant des durées *D_décroissante* donne les meilleurs résultats. Dans le tableau 4.I où nous notons en caractère gras les meilleurs résultats, nous pouvons obser-

ver le pourcentage d'instances pour lesquelles chaque heuristique est la meilleure. Nous pouvons remarquer également que l'heuristique *D+DTL_décroissante* obtient les mêmes résultats. Ceci s'explique par le fait que la génération des time lags fait que ce paramètre est en corrélation forte avec la durée.

Ordre	D	DTL	D+DTL	D/DTL	Lexico
Décroissant	**55**	51	**55**	29	20
Croissant	5	12	5	22	

Tableau 4.I – Comparaison des heuristiques

Comparaison des variantes proposées

Nous avons essayé toutes les combinaisons de paramétrages : positionnement des divergences (*top-first*, nommé *TF* par la suite, ou *bottom-first*), modes d'incrémentations des poids des jobs (*mode A*, *mode B*, *mode C*, ou 0) et modes de comptages (*Max* ou *Somme*).

	ft06	*la01-05*	*la06-10*	*la11-15*	*la16-20*	*Total*
TF	1	21	19	19	16	**76**
A-Sum	5	6	3	4	6	24
A-Max	2	4	1	5	6	18
B-Sum	*3*	*6*	*5*	*5*	*8*	***27***
B-Max	*5*	*8*	*0*	7	7	**27**
C-Sum	3	8	0	6	7	24
C-Max	3	10	1	6	7	27

Tableau 4.II – Comparaison des variantes proposées

Le tableau 4.II présente la comparaison des différentes variantes proposées sur les instances $ft06$ et $\{laX\}_{X=1..20}$. Sur ce tableau où nous retenons les meilleurs résultats obtenus, *TF* désigne la version de CDS sans poids sur les jobs qui diverge en haut de

l'arbre de recherche d'abord. *X-Sum*, respectivement *X-Max*, désignent les *cas A*, *B*, ou *C* de l'incrémentation des jobs, présentés au préalable, associés aux mode de comptage *Sum* ou *Max* respectivement qui divergent en haut de l'arbre de recherche d'abord également.

Les résultats montrent que la méthode *TF* sans poids sur les jobs parvient aux meilleurs résultats. Parmi les cas de comptage considérés, le *cas B* qu'il soit associé au mode de comptage *Sum* ou *Max* semble être meilleur que les autres cas d'incrémentation étudiés. Dans la suite des tests, nous nous concentrons sur les meilleures variantes testées et donc *TF* ainsi que le *cas B*.

Comparaison *vs.* les meilleurs résultats connus

Les meilleures solutions connues (*BKS*) sont en général partagées entre l'algorithme mémétique de [Caumond 2008], ILOG-Scheduler et nos propositions, sans grande régularité. Nous pouvons remarquer que pour les problèmes de type *no-wait* (*i.e.,* avec $\alpha = 0$), [Caumond 2008] obtient les meilleurs résultats. Pour les autres instances, ILOG-Scheduler obtient les meilleurs résultats sauf pour les instances présentées sur le tableau 4.III où nos propositions se démarquent de manière positive.

Nous pouvons toutefois remarquer que tout d'abord, sur les instances de flowshop, nous pouvons dire que les résultats obtenus par nos propositions sont globalement moins bons que les meilleures solutions connues (*BKS*). Les écarts aux solutions optimales varient entre 0 et 15%.

Pour les jobshops, les résultats sont plus partagés. Les BKS sont ainsi alternativement trouvées par l'algorithme mémétique de [Caumond 2008], ILOG-Scheduler et nos propositions, sans aucune régularité. Ce que nous pouvons néanmoins affirmer est que pour les problèmes de type no-wait (time lags maximaux égaux à zéro), l'algorithme mémétique obtient les meilleurs résultats. Pour les autres instances, ILOG-Scheduler fournit généralement les meilleurs résultats, à l'exception des instances répertoriées dans le tableau 4.III pour lesquelles nos propositions fournissent les meilleures performances. Dans les trois dernières colonnes de ce tableau, nous présentons les résultats obtenus par le meilleur paramétrage, c'est-à-dire en considérant *TF*, *B-Sum* et *B-Max*. Nous rap-

pelons que *TF* désigne la version de CDS sans poids avec des divergences effectuées prioritairement en haut de l'arbre, *B-Sum* (respectivement *B-Max*) fait référence à un mode de pondération de type *cas B* associé à un mode de comptage *Sum* (resp. *Max*). Cas B semble en effet être meilleur que les autres mode de pondération des jobs sur les instances considérés. Dans le tableau, les caractères gras représentent les meilleurs résultats.

Instance	*TLmax*	*ILOG-Scheduler*	*TF*	*B-Sum*	*B-Max*
la11	0.25	2058	**1861**	1965	1965
	0.5	1945	**1874**	**1874**	**1874**
la12	0.25	1710	1682	1671	**1656**
la13	0.25	1906	**1897**	1892	1892
	0.5	1804	**1787**	1808	1808
la14	0.25	2143	**1823**	2042	2042
	0.5	2067	1964	**1953**	**1953**
	1	1976	1772	**1762**	**1762**
	2	1976	**1612**	1660	1660
	3	1695	1567	**1542**	**1542**
	5	1695	**1452**	1477	1477
la15	0.25	2371	2084	**2043**	**2043**
	0.5	2217	2118	**1910**	**1910**
la17	0.25	1455	**1410**	1427	1460

Tableau 4.III – Résultats obtenus sur quelques instances

Variantes à divergences limitées par profondeur

Nous avons testé également une variante de CDS avec divergences limitées par la profondeur, de telle sorte que l'on ne diverge plus en bas de l'arborescence à partir d'une certaine profondeur fixée au préalable. Nous avons fait varier les valeurs de cette profondeur. Les résultats obtenus sont très encourageants. Dans le tableau 4.IV, nous présentons ceux obtenus avec une profondeur égale à 0.5 de la hauteur de l'arbre de recherche. Nous pouvons observer que la variante *B-Max* se démarque. L'étude faite sur ce type de variantes est à affiner surtout que d'une profondeur à une autre, la différence de la qualité des résultats obtenus est palpable.

	TF	*B-Sum*	*B-Max*
ft06	3	3	5
la01-05	7	20	23
la06-10	16	11	15
la11-15	9	24	25
la16-20	12	19	16
ft10	1	6	6
car5-8	8	15	23
Total	56	98	**113**

Tableau 4.IV – Comparaison des variantes à divergences limitées par profondeur (*profondeur 0.5*)

4.5 Conclusion

Dans ce chapitre, nous proposons des mécanismes pour exploiter le principe des méthodes à base de divergences pour la résolution de problèmes d'ordonnancement de types jobshop et flowshop avec contraintes de time lags. Nous adaptons ainsi la méthode Climbing Discrepancy Search (CDS) en étudiant plusieurs paramètres liés à la résolution comme la position des divergences, l'heuristique de génération de la solution initiale et des techniques d'apprentissage basées sur des poids associés aux jobs. Les variantes proposées ont été testées sur des benchmarks de la littérature. Les résultats obtenus sont satisfaisants pour certaines instances. Ceci nous incite désormais à étudier l'impact de CDS associée à des techniques classiques en ordonnancement comme les heuristiques d'insertion pour la détermination de bornes supérieures et le réglage de règles de propagation pour les bornes inférieures. La génération de benchmarks plus variés, de telle sorte que la définition du time lag ne soit pas nécessairement corrélée à la durée du job, est à envisager.

CONCLUSION GÉNÉRALE & PERSPECTIVES

Dans ce travail de thèse, nous avons étudié les procédures de recherche arborescente à base de divergences pour la conception de méthodes complètes dédiées à la satisfaction de contraintes ainsi que pour la résolution de problèmes d'optimisation.

Dans une première partie, nous avons présenté des techniques permettant l'amélioration de méthodes de résolution de problèmes de satisfaction de contraintes en particulier celles basées sur les divergences (méthodes de la famille LDS). Nous proposons des mécanismes qui profitent des échecs rencontrés au cours de la résolution. Ces mécanismes se concrétisent par la pondération de poids associés aux variables ainsi qu'aux valeurs. Ces poids sont exploités dans les heuristiques d'ordre d'instanciation pour orienter les futurs choix des heuristiques. Particulièrement pour les méthodes à divergences, nous proposons des techniques de restriction des divergences superflues et différents types de comptage des divergences. Une variante qui combine quelques-uns de ces mécanismes a été plus spécifiquement étudiée. Nous considérons également une amélioration de cette variante par une exploitation simple des no-goods.

La plupart de ces mécanismes ont été évalués sur des benchmarks de la littérature concernant des problèmes de séquençage de voitures nécessitant l'installation d'options (*car sequencing*) ainsi que des problèmes générés aléatoirement. Les résultats expérimentaux obtenus attestent de l'efficacité de certaines de ces techniques. Nous évaluons ainsi l'impact des modes de comptage des divergences (binaire, non-binaire ou mixte) et la performance des heuristiques d'instanciation proposées basées sur l'utilisation de poids sur les variables et les valeurs. Concernant les problèmes de car sequencing, nous concluons que la méthode LDS avec une heuristique qui associe des poids aux variables (les conflits étant réglés par un ordre lexicographique) obtient les meilleurs résultats. Concernant les problèmes aléatoires, nous constatons que la méthode de maintien de consistance d'arc (MAC) associée à une heuristique qui ordonne les variables selon le rapport des tailles de leur domaines et des poids associés aux variables donne globa-

lement de meilleurs résultats que ceux obtenus par MAC avec une heuristique qui ordonne les variables selon le rapport des tailles de leur domaines et des poids associés aux contraintes. En comparant différentes heuristiques (rapport des tailles des domaines et des poids associés aux contraintes et rapport des tailles des domaines et des poids associés aux variables) dans une méthode de type LDS (intégrant une combinaison des mécanismes que nous proposons, méthode que nous avons nommée YIELDS), considérés en comptage binaire et non-binaire, nous obtenons que l'heuristique qui ordonne les variables selon le rapport des tailles des domaines et des poids associés aux variables que nous proposons donne des résultats meilleurs, surtout en comptage non-binaire.

Dans une seconde partie, nous proposons des mécanismes pour exploiter le principe des méthodes à base de divergences pour la résolution de problèmes d'ordonnancement de types jobshop et flowshop avec contraintes de time lags. Nous adaptons ainsi la méthode Climbing Discrepancy Search (CDS) en étudiant plusieurs paramètres liés à la résolution comme la position des divergences, l'heuristique de génération de la solution initiale et des techniques d'apprentissage basées sur des poids associés aux jobs. Pour la position des divergences, nous considérons des divergences en haut d'abord et en bas d'abord. Pour la génération de la solution initiale, nous étudions les heuristiques basées sur les durées des jobs ainsi que sur les durées des time lags et de leurs combinaisons par une somme ou un rapport. Les techniques d'apprentissage sont concrétisées par des poids associés aux jobs. Un poids est incrémenté à chaque fois que l'on rencontre un échec d'insertion de ce job. Plusieurs modes de pondération ont été considérés.

Toutes les combinaisons possibles des mécanismes proposés ont été envisagées. Les variantes proposées ont été testées sur des benchmarks de la littérature. Les résultats des tests concernant l'heuristique de génération de la solution initiale montrent que l'heuristique qui se base sur un ordre décroissant des durées des jobs donne les meilleurs résultats. La comparaison des différentes variantes proposées montrent que la version de CDS sans poids sur les jobs qui diverge en haut de l'arbre de recherche d'abord parvient aux meilleurs résultats, tout comme la version qui adopte le cas de comptage binaire

sur les machines pour l'incrémentation des poids sur les jobs, que celle-ci soit associée à une récapitulation des poids par leur somme ou leur maximum. Les meilleures solutions connues (*BKS*) sont partagées entre l'algorithme mémétique de Caumond, ILOG-Scheduler et nos propositions. Sur les instances de flowshop, nous pouvons dire que les résultats obtenus par nos propositions sont globalement moins bons que les BKS. Pour les jobshops, les résultats obtenus sont satisfaisants pour certaines instances et nous améliorons les meilleurs résultats connus.

Les résultats obtenus dans les deux parties de cette thèse nous incitent désormais à poursuivre l'étude de certaines pistes.

Une piste à considérer à court terme, valable pour les problèmes de satisfaction ainsi que les problèmes d'optimisation, est la considération de benchmarks plus variés et l'adaptation des techniques proposées à la nature des problèmes étudiés. En effet, de telles adaptations pourront contribuer à l'amélioration des performances des méthodes considérées. La considération d'autres heuristiques profitant des mécanismes proposés est également à considérer (autres modes de comptage, autres modes d'incrémentation des poids, ...). Pour les méthodes de résolution pour l'optimisation, une étude de l'impact de CDS associée à de nouvelles heuristiques d'insertion pour la détermination de bonnes bornes supérieures et le réglage de règles de propagation pour les bornes inférieures pourraient facilement se révéler bénéfiques.

Pour les pistes à envisager à moyen voire long terme, nous pouvons intégrer une analyse poussée des no-goods et profiter des déductions qui s'imposent pour doper les performances des méthodes considérées. Les CSP ont été considérés dans ce travail uniquement dans un contexte de satisfaction, on peut envisager de joindre CSP et optimisation et profiter des travaux déjà effectuées en CSOP (Constraint Satisfaction and Optimization Problems) et WCSP (Weighted Constraint Satisfaction and Optimization Problems).

BIBLIOGRAPHIE

[Achlioptas 2001] D. Achlioptas, M. S. O. Molloy, L. M. Kirousis, Y. C. Stamatiou, E. Kranakis et D. Krizanc. *Random Constraint Satisfaction : A more Accurate Picture*. Constraints, vol. 6, pages 329–344, 2001.

[Aggoun 1993] A. Aggoun et N. Beldiceanu. *Extending CHIP in Order to Slove Complex Scheduling and Placement Problems*. Mathematical and Computer Modelling, vol. 17, no. 7, pages 57–73, 1993.

[Apt 2000] K. R. Apt. *Principles of Constraint Programming*, 2000.

[Baptiste 2000] P. Baptiste. *Programmation par contraintes, recherche opérationnelle et ordonnancement*. In 3ème Congrès de la Société Française de Recherche Opérationnelle et d'Aide à la Décision ROADEF 2000, Nantes, France, Janvier 2000.

[Beck 2000] J. Beck et L. Perron. *Discrepancy-bounded depth first search*. In Proceedings of the 2nd International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'00), Paderborn, Germany, March 2000.

[Beek 1995] P. Van Beek et R. Dechter. *On the minimality and global consistency of row-convex constraint networks*. Communications of the ACM, vol. 42, no. 3, pages 543–561, 1995.

[Beek 2006] P. Van Beek, F. Rossi et T. Walsh (editors). Handbook of constraint programming. Elsevier, 2006.

[Beldiceanu 1994] N. Beldiceanu et E. Contjean. *Introducing global constraints in CHIP*. Mathematical and Computer Modelling, vol. 12, pages 97–123, 1994.

[Berlandier 1995] P. Berlandier. *Improving domain filtering using restricted path consistency*. In Proceedings of the 11th Conference on Artificial Intelligence

for Applications (CAIA'95), IEEE Computer Society, page 32, Washington, DC, USA, 1995.

[Bessière 1993] C. Bessière et M. O. Cordier. *Arc-consistency and arc-consistency again*. In Proceedings of the National Conference on Artificial Intelligence (AAAI'93), pages 108–113, Washington, DC, USA, 1993.

[Bessière 1994] C. Bessière. *Arc-consistency and arc-consistency again*. Artificial Intelligence, vol. 65, pages 179–190, 1994.

[Bessière 1995] C. Bessière, E. C. Freuder et J. C. Régin. *Using inference to reduce arc consistency computation*. In Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI'95), pages 592–598, Montréal, Québec, Canada, August 1995.

[Bessière 1997] C. Bessière et J. C. Régin. *Arc consistency for general constraint networks : preliminary results*. In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'97), pages 398–404, Nagoya, Japan, 1997.

[Bessière 1999] C. Bessière et J. C. Régin. *Enforcing arc consistency on global constraints by solving subproblems on the fly*. In Proceedings of Principles and Practice of Constraint Programming (CP'99), Springer-Verlag, pages 103–117, London, UK, 1999.

[Bessière 2001] C. Bessière et J.-C. Régin. *Refining the basic constraint propagation algorithm*. In Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01), pages 309–315, Seattle, Washington, USA, August 2001.

[Bessière 2005] C. Bessière, J.C. Régin, R.H.C. Yap et Y. Zhang. *An optimal coarse-grained arc consistency algorithm*. Artificial Intelligence, 2005.

[Bessière 2006] C. Bessière, E. Hebrard, B. Hnich, Z. Kiziltan et T. Walsh. *Filtering algorithms for the nvalue constraint*. Constraints, vol. 11, no. 4, pages 271–293, 2006.

[Bessière 2010] C. Bessière, S. Cardon, R. Debruyne et C. Lecoutre. *Efficient algorithms for singleton arc consistency*. Constraints, 2010.

[Boivin 2005] S. Boivin, M. Gravel, M. Krajecki et C. Gagné. *Résolution du problème de car-sequencing à l'aide d'une approche de type FC*. In Premières Journées Francophones de la Programmation par Contraintes (JFPC'05), pages 11–20, Lens, France, Juin 2005.

[Boussemart 2004a] F. Boussemart, F. Hemery, C. Lecoutre et L. Sais. *Boosting systematic search by weighting constraints*. In Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'04), pages 146–150, Valencia, Spain, August 2004.

[Boussemart 2004b] F. Boussemart, F. Hemery, C. Lecoutre et L. Sais. *Heuristiques de choix de variables dirigées par les conflits*. In Proceedings of JNPC'04, pages 91–105, Angers, France, June 2004.

[Cambazard 2006] H. Cambazard. *Résolution de problèmes combinatoires par des approches fondées sur la notion d'explication*. Thèse de doctorat, Université de Nantes, Nantes, 2006.

[Caseau 1996] Y. Caseau et F. Laburthe. *CLAIRE : A parametric Tool to Generate C++ Code for Problem Solving*. Rapport technique, Bouygues, Direction Scientifique., 1996.

[Caumond 2004] A. Caumond, M. Gourgand, P. Lacomme et N. Tchernev. *Métaheuristiques pour le problème de jobshop avec time lags :* $Jm|l_{i,sj(i)}|C_{\max}$. In 5ème confèrence Francophone de MOdélisation et SIMulation (MOSIM'04). Modélisation et simulation pour l'analyse et l'optimisation des systèmes industriels et logistiques, pages 939–946, Nantes, France, 2004.

[Caumond 2008] A. Caumond, P. Lacomme et N. Tchernev. *A memetic algorithm for the job-shop with time-lags*. vol. 35, pages 2331–2356, 2008.

[Chmeiss 1998] A. Chmeiss et P. Jégou. *Efficient Path-Consistency Propagation*. International Journal for Artificial Intelligence Tools, vol. 7, no. 2, pages 121–142, 1998.

[Colmerauer 1982] A. Colmerauer, H. Kanoui et M. Van Caneghem. *Prolog, bases théoriques et développements actuels*. Technique et science informatiques, Hermès, vol. 2, no. 4, pages 29–127, 1982.

[CSPLib] CSPLib. `http://csplib.org`.

[Debruyne 1997] R. Debruyne et C. Bessière. *Some practicable filtering techniques for the constraint satisfaction problem*. In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'97), pages 412–417, Nagoya, Japan, 1997.

[Debruyne 1998] R. Debruyne. *Consistances locales pour les problèmes de satisfaction de contraintes de grande taille*. Thèse de doctorat, Université de Montpellier II, 1998.

[Dechter 1989] R. Dechter et I. Meiri. *Experimental evaluation of preprocessing techniques in constraint satisfaction problems*. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI'89), pages 271–277, Detroit, Michigan, USA, 1989.

[Dechter 2003] R. Dechter. Constraint processing. Morgan Kaufmann, San Francisco, May 2003.

[Dell'amico 1996] M. Dell'amico. *Shop problems with two machines and time lags*. Operations Research, vol. 44, no. 5, pages 777–787, 1996.

[Deville 1991] Y. Deville et P. Van Hentenryck. *An efficient arc consistency algorithm for a class of CSP problems*. In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'91), pages 325–330, 1991.

[Dincbas 1988] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf et F. Berthier. *The Constraint Logic Programming Language CHIP*. In In Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88), pages 693–702, Tokyo, 1988.

[Esquirol 1995] P. Esquirol, P. Lopez, H. Fargier et T. Schiex. *Constraint programming*. Belgian Journal of Operations Research, Special Issue Constraint Programming (JORBEL), vol. 35, no. 2, pages 5–36, 1995.

[Freuder 1978] E. C. Freuder. *Synthesizing constraint expressions*. Communications of the ACM, vol. 21, no. 11, pages 958–966, 1978.

[Freuder 1982] E. C. Freuder. *A sufficient condition for backtrack-free search*. Communications of the ACM, vol. 29, no. 1, pages 24–32, 1982.

[Freuder 1996] E. C. Freuder et C. D. Elfe. *Neighborhood inverse consistency preprocessing*. In Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI'96), volume 1, pages 202–208, 1996.

[Freuder 1997] E. C. Freuder. *In Pursuit of the Holy Grail*. Constraints, pages 57–61, 1997.

[Frost 1996] D. H. Frost, C. Bessière, R. Dechter et J.C. Régin. *Random uniform CSP generators. 1996*, 1996. `http://www.lirmm.fr/~bessiere/generator.html`.

[Gacias 2008] B. Gacias, C. Artigues et P. Lopez. *Tree and local search for parallel machine scheduling problems with precedence constraints and setup times*. In Proceedings of the 11th International Workshop on Project Management and Scheduling (PMS'08), Istanbul, Turkey, April 2008.

[Grimes 2006] D. Grimes et R. J. Wallace. *Learning from failures in constraint satisfaction search*. In AAAI Workshop on Learning for Search, Boston, Massachusetts, USA, July 2006.

[Hansen 2001] P. Hansen et N. Mladenovic. *Variable neighborhood search : principles and applications (invited review)*. European Journal of Operational Research, vol. 130, pages 130–467, 2001.

[Haralick 1980] R. Haralick et G. Elliot. *Increasing tree search efficiency for constraint satisfaction problems*. Artificial Intelligence, vol. 14, pages 263–313, 1980.

[Harvey 1995a] W. D. Harvey. *Nonsystematic Backtracking Search*. PhD thesis, Stanford University, Stanford, CA, 1995.

[Harvey 1995b] W. D. Harvey et M. L. Ginsberg. *Limited Discrepancy Search*. In Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI'95), volume 1, pages 607–615, Montréal, Québec, Canada, August 1995.

[Hentenryck 1992a] P. Van Hentenryck, Y. Deville et C. M. Teng. *A generic arc-consistency algorithm and its specializations*. Artificial Intelligence, vol. 57, no. 2-3, pages 291–321, 1992.

[Hentenryck 1992b] P. Van Hentenryck, H. Simonis et M. Dincbas. *Constraint Satisfaction using Constraint Logic Programming*. Artificial Intelligence, vol. 58, pages 113–159, 1992.

[Hmida 2007] A. Ben Hmida, M. J. Huguet, P. Lopez et M. Haouari. *Climbing Discrepancy Search for solving the hybrid Flow shop*. European Journal of Industrial Engineering, vol. 1, no. 2, pages 223–243, July 2007.

[Hmida 2010] A. Ben Hmida, M. Haouari, M. J. Huguet et P. Lopez. *Discrepancy search for the flexible job shop problem*. Computers and Operation Research, 2010. to appear.

[Hooker 2000] J. Hooker. Logic-based methods for optimization : combining optimization and constraint satisfaction. Wiley, John and Sons, Inc. (first published 2000), Wiley, 2000.

[Huguet 2004] M. J. Huguet, P. Lopez et A. Ben Hmida. *A Limited Discrepancy Search method for solving disjunctive scheduling problems with resource flexibility*. In Ninth International Workshop on Project Management and Scheduling (PMS'04), Nancy, (France), pages 299–302, April, 2004.

[Huguet 2010] M. J. Huguet, C. Artigues, M. Dugas et P. Lopez. *Generalized Constraint Propagation for Solving Job Shop Problems with time lags*. In Proceedings of the 12th International Workshop on Project Management and Scheduling (PMS'10), pages 239–242, Tours, France, April 2010.

[Jaffar 1994] J. Jaffar et M. Maher. *Constraint Logic Programming : A Survey*. In Journal of Logic Programming, volume 19/20, pages 503–581, 1994.

[Karoui 2005] W. Karoui, M. J. Huguet, P. Lopez et W. Naanaa. *Amélioration par apprentissage de la recherche à divergences limitées*. In Premières Journées Francophones de la Programmation par Contraintes (JFPC'05), pages 109–118, Lens, France, Juin 2005.

[Karoui 2006] W. Karoui, M. J. Huguet, P. Lopez et W. Naanaa. *Amélioration par apprentissage d'une méthode de recherche arborescente*. In 7ème Congrès de la Société Française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF'2006), Lille, France, Février 2006.

[Karoui 2007a] W. Karoui, M. J. Huguet, P. Lopez et W. Naanaa. *Apport de MDS à la résolution de CSP : application aux problèmes de Job Shop et de carrés latins*. In Conférence scientifique conjointe en Recherche Opérationnelle et Aide à la Décision FRANCORO V (ROADEF'2007), Grenoble, France, Février 2007.

[Karoui 2007b] W. Karoui, M. J. Huguet, P. Lopez et W. Naanaa. *YIELDS : A Yet Improved Limited Discrepancy Search for CSPs*. In Proceedings of the 4th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'07), volume 4510

of *Lecture Notes in Computer Science*, pages 99–111, Brussels, Belgium, May 2007. Springer–Verlag.

[Karoui 2009] W. Karoui, M. J. Huguet et P. Lopez. *Impact des modes de comptage sur les méthodes à base de divergences*. In Conférence scientifique en Recherche Opérationnelle et d'Aide à la Décision (ROADEF'2009), Nancy, France, Février 2009.

[Karoui 2010a] W. Karoui, M. J. Huguet, P. Lopez et M. Haouari. *Climbing discrepancy search for flowshop and jobshop with time lags*. In Proceedings of the 4th International Symposium on Combinatorial Optimization (ISCO'10), volume 1140 of *Electronic Notes in Discrete Mathematics*, Hammamet, Tunisie, Mars 2010. Elsevier.

[Karoui 2010b] W. Karoui, M. J. Huguet, P. Lopez et M. Haouari. *Heuristiques à divergence limitée pour les problèmes d'ordonnancement avec contraintes de délais*. In Conférence scientifique en Recherche Opérationnelle et d'Aide à la Décision (ROADEF'2010), Toulouse, France, Février 2010.

[Karoui 2010c] W. Karoui, M. J. Huguet, P. Lopez et M. Haouari. *Limited discrepancy search for scheduling problems with time lags*. In 12th International Workshop on Project Management and Scheduling (PMS'10), pages 273–276, Tours, France, Avril 2010.

[Karoui 2010d] W. Karoui, M. J. Huguet, P. Lopez et M. Haouari. *Méthodes de recherche à divergence limitée pour les problèmes d'ordonnancement avec contraintes de délais*. In 8ème conférence internationale de Modélisation et SIMulation (MOSIM'10), Hammamet, Tunisie, Mai 2010.

[Korf 1996] R.E. Korf. *Improved limited Discrepancy Search*. In Proceedings of the 13th National Conference on Artificial Intelligence (AAAI'96) and the 8th Innovative Applications of Artificial Intelligence Conference (IAAI'96), pages 286–291, Portland, Oregon, USA, August 1996.

[Laburthe 1998] F. Laburthe, P. Savéant, S. de Givry et J. Jaurdan. *Eclair, a library of constraints over finite domains*. Rapport technique, ATS 98-2, Thomson CSF, Corporate Research Lab., 1998.

[Lauriere 1978] J. L. Lauriere. *ALICE : A Language and a Program for Solving Combinatorial Problems*. Artificial Intelligence, vol. 10, pages 29–127, 1978.

[Lecoutre 2006] C. Lecoutre, L. Sais, S. Tabary et V. Vidal. *Last Conflict based Reasoning*. In Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06), pages 133–137, Trento, Italy, August 2006.

[Lecoutre 2007a] C. Lecoutre et F. Hemery. *A study of residual supports in arc consistency*. In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'07), pages 125–130, 2007.

[Lecoutre 2007b] C. Lecoutre, L. Sais et J. Vion. *Using SAT Encodings to derive CSP Value Ordering Heuristics*. Journal on Satisfiability, Boolean Modeling and Computation, vol. 1, pages 69–186, 2007.

[Lecoutre 2008] C. Lecoutre, C. Likitvivatanavong, S. Shannon, R. Yap et Y. Zhang. *Maintaining Arc Consistency with Multiple Residues*. Constraint Programming Letters (CPL), vol. 2, pages 3–19, 2008.

[Levasseur 2007] N. Levasseur, P. Boizumault et S. Loudni. *A Value Ordering Heuristic for Weighted CSPs*. In Proceedings of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI'07), pages 259–262, Patras, Greece, October 2007.

[Likitvivatanavong 2004] C. Likitvivatanavong, Y. Zhang, J. Bowen, et E. C. Freuder. *Arc-consistency in MAC : A new perspective*. In Proceedings First International Workshop on Constraint Propagation and Implementation, Toronto, Canada, 2004.

[Mackworth 1977] A. K. Mackworth. *Consistency in networks of relations*. Artificial Intelligence, vol. 8, pages 99–118, 1977.

[Mackworth 1985] A. K. Mackworth et E. C. Freuder. *The complexity of some polynomial network consistency algorithms for constraint satisfaction problems*. Artificial Intelligence, vol. 25, pages 65–74, 1985.

[Mehta 2005] D. Mehta et M. R. C. Van Dongen. *Reducing checks and revisions in coarse-grained MAC algorithms*. In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'05), pages 236–241, 2005.

[Meseguer 1998] P. Meseguer et T. Walsh. *Interleaved and Discrepancy Based Search*. In Proceedings of the 13 European Conference on Artificial Intelligence (ECAI'98), pages 239–243, Brighton, UK, 1998. John, Wiley and Sons Inc.

[Milano 2002] M. Milano et A. Roli. *On the relation between complete and incomplete search : an informal discussion*. In Proceedings of the 4th International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'02), Le Croisic, France, 2002.

[Mitten 1958] L. G. Mitten. *Sequencing n jobs on two machines with arbitrary time lags*. Management Science, vol. 5, pages 293–298, 1958.

[Mohr 1986] R. Mohr et T. C. Henderson. *Arc and path consistency revised*. Artificial Intelligence, vol. 28, pages 225–233, 1986.

[Mohr 1987] R. Mohr. *A correct path consistency algorithm and an optimal generalized arc consistency algorithm*. Rapport technique, CRIN, 87-R-030, 1987.

[Mohr 1988] R. Mohr et G. Masini. *Good old discrete relaxation*. In Proceedings of the European Conference on Artificial Intelligence (ECAI'88), pages 651–656, 1988.

[Montanari 1974] U. Montanari. *Networks of constraints : Fundamental properties and applications to picture processing*. In Information Science, volume 7(2), pages 95–132, 1974.

[Néron 2008] E. Néron, F. Tercinet et F. Sourd. *Search tree based approaches for parallel machine scheduling*. Computers and Operation Research, vol. 35, no. 4, pages 1127–1137, 2008.

[Nowicki 1996] E. Nowicki et C. Smutnicki. *A fast taboo serach algorithm for the job-shop problem*. Management Science, vol. 42(6), pages 797–813, 1996.

[Perlin 1992] M. Perlin. *Arc consistency for factorable relations*. Artificial Intelligence, vol. 53, pages 329–342, 1992.

[Prcovic 2002] N. Prcovic. *Quelques variantes de LDS*. In Proceedings of the VIIIe Journées Nationales sur les Problèmes NP-Complets (JNPC'02), pages 195–208, Nice, 2002.

[Prosser 1993] P. Prosser. *Hybrid algorithms for the constraint satisfaction problem*. Computational Intelligence, vol. 9, no. 3, pages 268–299, August 1993.

[Puget 1995] J. F. Puget et M. Leconte. *Beyond the Glass Box : Constraints as Objets*. In Twelfth International Symposium on Logic Programming, Portland, Oregon, 1995.

[Refalo 2004] P. Refalo. *Impact-based search strategies for constraint programming*. In Proceedings of Principles and Practice of Constraint Programming (CP'04), LNCS 3258, Springer, pages 557–571, Toronto, Canada, September 2004.

[Régin 1994] J. C. Régin. *A filtering algorithm for constraints of difference in CSPs*. In Proceedings of the twelfth National Conference on Artificial Intelligence (AAAI'94). American Association for Artificial Intelligence, volume 1, pages 362–367, Menlo Park, CA, USA, 1994.

[Sabin 1994] D. Sabin et E. C. Freuder. *Contradicting conventional wisdom in constraint satisfaction*. In Proceedings of the 2nd International Workshop on Principles and Practices of Constraint Programming (PPCP'94), LNCS 874, Springer, pages 10–20, Rosario, Orcas Island, Washington, USA, May 1994.

[Smith 1997] B. Smith. *Succeed-first or Fail-first : A Case Study in Variable and Value Ordering Heuristics*. In Proceedings of the 3rd Conference on the Practical Applications of Constraint Technology (PACT'97), pages 321–330, London, UK, April 1997.

[Solnon 2003] C. Solnon. *Cours de Programmation par contraintes (e-miage)*, 2003. http://www710.univ-lyon1.fr/~csolnon/Site-PPC/e-miage--ppc-som.htm.

[Solnon 2008] C. Solnon, V. D. Cung, A. Nguyen et C. Artigues. *The car sequencing problem : overview of state-of-the-art methods and industrial case-study of the ROADEF'05 challenge problem*. European Journal of Operational Research, vol. 191, no. 3, pages 912–927, 2008.

[Sutherland 1963] I. Sutherland. *Sketchpad : A Man Machine Graphical Communication System*. PhD thesis, Massachusetts Institute of Technology, Massachusetts, 1963.

[Tsang 1993] E. Tsang. Foundations of constraint satisfaction. Academic Press Ltd, London, August 1993.

[Walsh 1997] T. Walsh. *Depth-bounded Discrepancy Search*. In Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97), volume 2, pages 1388–1395, Nagoya, Japan, August 1997.

[Xu 2007] K. Xu, F. Boussemart, F. Hemery et C. Lecoutre. *Random Constraint satisfaction : Easy generation of hard (satisfiable) instances*. Artificial Intelligence, vol. 171, pages 514–534, 2007.

[Zhang 2001] Y. Zhang et R. H. C. Yap. *Making AC-3 an optimal algorithm*. In Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01), pages 316–321, Seattle, Washington, USA, August 2001.

Résumé

Le formalisme « Problème de Satisfaction de Contraintes » (ou CSP pour *Constraint Satisfaction Problem*) peut être considéré comme un langage de représentation formelle qui couvre l'ensemble des problèmes dont la modélisation fait intervenir des contraintes. L'intérêt de ce formalisme réside dans l'exploitation de la généricité d'algorithmes de résolution puissants mais également dans la performance d'algorithmes dédiés à des problèmes particuliers.

Dans ce travail de thèse, nous étudions la résolution de CSP par des méthodes de recherche arborescente basées sur la notion de « divergence » (une divergence est relative à la contradiction d'une décision proposée par une heuristique de référence). Dans ce cadre, nous proposons de nouveaux mécanismes d'amélioration des méthodes de recherche générales qui exploitent les échecs rencontrés pendant la résolution, en adoptant des heuristiques de pondération des variables et des valeurs. Nous proposons également d'autres techniques spécifiques aux méthodes à base de divergences qui conditionnent l'exploration de l'arbre de recherche développé, notamment la restriction des divergences, les différents modes de comptage ainsi que le positionnement des divergences. Ces propositions sont validées par des expérimentations numériques menées sur des problèmes de satisfaction de contraintes réels et aléatoires. Des comparaisons sont effectuées entre variantes de méthodes à divergences intégrant différentes combinaisons des améliorations et d'autres méthodes connues pour leur performance.

Dans une seconde partie, nous étendons nos propositions à un contexte d'optimisation en considérant la résolution de problèmes d'ordonnancement avec contraintes de délais (*time lags*). Nous traitons l'adaptation d'une méthode de « recherche par montée de divergences » (*Climbing Discrepancy Search*) pour la résolution de ces problèmes. Nous validons les performances de certaines variantes de cette méthode intégrant les mécanismes proposés dans ce travail sur des problèmes-test de la littérature.

Mots-clés :

Problèmes de satisfaction de contraintes, recherche arborescente, recherche à divergences, heuristiques, ordonnancement.

Abstract

The CSP (Constraint Satisfaction Problem) formalism can be considered as a simple example of a formal representation language covering all problems including constraints. The advantage of this formalism consists in the fact that it allows powerful general-purpose algorithms as much as useful specific algorithms.

In this PhD thesis, we study several tree search methods for solving CSPs and focus on ones based on the discrepancy concept (a discrepancy is a deviation from the first choice of the heuristic). In this context, we propose improving mechanisms for general methods. These mechanisms take benefits from conflicts and guide the search by weighting the variables and the values. We propose also special mechanisms for methods based on discrepancies as the discrepancies restriction, the discrepancies counting, and the discrepancies positions. All propositions are validated by experiments done on real and random CSPs. We compare variants of methods based on discrepancies integrating several combinations of improvements and other methods known for their efficiency.

In a second part, we extend our propositions to an optimisation context considering scheduling problems with time lags. In this purpose, we adapt a discrepancy-based method, Climbing Discrepancy Search, to solve these problems. Efficiency of some improved variants of this method is tested on known benchmarks.

Key words:

Constraint Satisfaction Problems, tree search, discrepancy search, heuristics, scheduling.

Title:

Discrepancy-based search for constraint satisfaction and combinatorial optimisation problems