Techniques d'ordonnancement d'atelier et de fournées basées sur la programmation par contraintes

Arnaud Malapert

UNIVERSITÉ DE NANTES
FACULTÉ DES SCIENCES ET DES TECHNIQUES

ECOLE DOCTORALE SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET MATHÉMATIQUES

Année 2011

N° attribué par la bibliothèque

Techniques d'ordonnancement d'atelier et de fournées basées sur la programmation par contraintes

---

THÈSE DE DOCTORAT

Discipline : Informatique

Spécialité : Informatique

Présentée
et soutenue publiquement par

Arnaud MALAPERT

Le 9 septembre 2011
à l'École Nationale Supérieure des Techniques Industrielles et des Mines de Nantes

Devant le jury ci-dessous :

Président	: Gilles Pesant, Professeur	École Polytechniques de Montréal
Rapporteur	: Christian Artigues, Chargé de recherche	LAAS-CNRS
Examinateurs	: Christelle Guéret, Maître-assistant	École des Mines de Nantes
	André Langevin, Professeur	École Polytechnique de Montréal

Directeur de thèse :
Narendra Jussien, Professeur, École des Mines de Nantes
Co-encadrant :
Louis-Martin Rousseau, Professeur, École Polytechnique de Montréal

Équipe d'accueil 1 : Équipe TASC, INRIA, LINA UMR CNRS 6241
Laboratoire d'accueil 1 : Département Informatique de l'École des Mines de Nantes
la Chantrerie, 4, rue Alfred Kastler, 44 307 Nantes

Équipe d'accueil 2 : Équipe Quosseça
Laboratoire d'accueil 2 : Département de mathématiques et génie industriel
École Polytechnique de Montréal
succursale Centre-ville, H3C 3A7 Montréal, Canada, C.P. 6079

ED : .....

Techniques d'ordonnancement d'atelier et de fournées basées sur la programmation par contraintes

Shop and batch scheduling with constraints

Arnaud MALAPERT

Université de Nantes

À mon père, Éric.

Nous sommes des nains juchés sur des épaules de géants. Nous voyons ainsi davantage et plus loin qu'eux, non parce que notre vue est plus aiguë ou notre taille plus haute, mais parce qu'ils nous portent en l'air et nous élèvent de toute leur hauteur gigantesque.

Bernard DE CHARTRES

— . . . La voie la plus féconde pour la recherche est celle qui implique la plus grande résistance à l'extérieur et la moindre à l'intérieur. L'échec doit être conçu comme une obligation de poursuivre ses efforts et de concentrer sa volonté. Si les efforts accomplis sont déjà considérables, l'échec ne doit être que plus joyeux ! C'est que notre barre a donné sur le couvercle de fer du trésor ! Et la victoire sur des difficultés accrues est d'autant plus appréciables que l'échec aura développé la puissance de l'exécutant à proportion des difficultés affrontées.
— Fameux ! Très fort ! approuva Nerjine du haut de son tas de rondins.
— Ce qui ne veut pas dire qu'il ne faut jamais renoncer à poursuivre un effort. Notre barre peut avoir heurté une pierre. Si on en est convaincu et que le milieu soit par trop hostile ou les moyens insuffisants, on est parfaitement en droit de renoncer au but. Mais il faut que cette renonciation soit très rigoureusement fondée.

Alexandre SOLJENITSYNE, *Le Premier Cercle*

Remerciements

Tous mes remerciements vont à Monsieur Gilles Pesant, professeur à l'École Polytechnique de Montréal, et à Monsieur Christian Artigues, chargé de recherche au LAAS-CNRS, qui m'ont fait l'honneur d'être les rapporteurs de ces travaux de thèse. J'adresse autant de remerciements à mes encadrants, Narendra, Louis-Martin, Christelle et André pour leur soutien, leurs conseils, leur patience et la liberté qu'ils m'ont laissé durant ces années.

Ce travail a été l'occasion de nombreuses rencontres enrichissantes de part et d'autre de l'Atlantique. À Nantes, je tiens à remercier les membres des équipes SLP et TASC pour leur disponibilité et la gentillesse dont ils ont fait preuve au début de ma thèse en répondant à toutes mes interrogations sans jamais perdre patience. À Montréal, je remercie tous les membres de l'équipe Quosséça et du CIRRELT qui ont facilité mon acclimatation et fait découvrir une nouvelle culture. À travers le monde, mais surtout à Cork, je remercie toutes les personnes avec qui j'ai eu l'occasion de collaborer sur des aspects scientifiques ou le développement de choco.

Je n'oublie pas non plus tous les professeurs, du collège à l'université, qui ont éveillé ma curiosité et mon goût pour les sciences. Je tiens également à remercier toutes les personnes qui m'ont donné ma chance en stage, mais aussi celles qui m'ont permis de concilier des études et un emploi.

Pour finir, je remercie mes parents qui ont toujours cru en moi, ma famille et mes amis pour les bons moments passés ensembles, mais aussi pour leur présence dans les moments difficiles. Enfin, cette expérience n'aurait pas été la même sans la présence et l'amour de ma compagne Marie dans toutes ces pérégrinations en y apportant son enthousiasme et sa joie de vivre. Je remercie notre fils Raphaël qui m'a insufflé l'élan pour clore cette étape de ma vie et nous entraîne vers de nouvelles aventures ...

Merci à toi lecteur de t'intéresser à cette modeste contribution.

Table des matières

Remerciements **iii**

Table des matières **v**

1 Introduction **1**
1.1 Objectifs . . . 2
1.2 Contribution . . . 3
1.3 Organisation du document . . . 4
1.4 Diffusion scientifique . . . 4

I Contexte de l'étude

2 Programmation par contraintes **9**
2.1 Modélisation d'un problème par des contraintes . . . 10
2.1.1 Problème de satisfaction de contraintes . . . 11
2.1.2 Problème d'optimisation sous contraintes . . . 12
2.1.3 Exemple : le problème des n reines . . . 12
2.2 Méthodes de résolution . . . 13
2.2.1 Algorithmes simples de recherche . . . 13
2.2.2 Filtrage et propagation des contraintes . . . 13
2.2.3 Contraintes globales . . . 15
2.2.4 Algorithmes avancés de recherche . . . 15
2.3 Stratégies de recherche . . . 16
2.3.1 Méthodes de séparation . . . 16
2.3.2 Heuristiques de sélection . . . 17
2.4 Procédures d'optimisation . . . 19
2.4.1 Procédure bottom-up . . . 20
2.4.2 Procédure top-down . . . 20
2.4.3 Procédure dichotomic-bottom-up . . . 21
2.4.4 Procédures incomplètes . . . 21
2.5 Solveurs de contraintes . . . 23
2.6 Conclusion . . . 24

3 Ordonnancement sous contraintes **25**
3.1 Tâches . . . 26
3.2 Contraintes temporelles . . . 27
3.2.1 Contraintes de précédence . . . 27
3.2.2 Contraintes de disponibilité et d'échéance . . . 27
3.2.3 Contraintes de disjonction . . . 27
3.2.4 Problèmes temporels . . . 28
3.3 Contraintes de partage de ressource . . . 28
3.3.1 Contrainte disjonctive . . . 29
3.3.2 Contrainte cumulative . . . 30

3.4 Modèle disjonctif . . . 31
3.5 Placement sous contraintes . . . 32
3.5.1 Placement en une dimension . . . 32
3.5.2 Placement en deux dimensions . . . 33
3.6 Conclusion . . . 34

4 Ordonnancement d'atelier 35
4.1 Définition des problèmes d'atelier . . . 35
4.1.1 Définition des problèmes de base . . . 36
4.1.2 Variantes et classification de Lawler . . . 36
4.1.3 Applications . . . 37
4.2 Résultats de complexité . . . 38
4.2.1 Problèmes à deux machines . . . 38
4.2.2 Problèmes à trois machines : la frontière . . . 38
4.2.3 Cas général . . . 38
4.2.4 Conclusion . . . 39
4.3 État de l'art . . . 39
4.3.1 Jeux d'instances . . . 39
4.3.2 Méthodes approchées . . . 39
4.3.3 Méthodes Exactes . . . 41
4.4 Orientation de nos travaux . . . 42

5 Ordonnancement d'une machine à traitement par fournées 43
5.1 Définition des problèmes de fournées . . . 43
5.1.1 Définition du problème de base . . . 44
5.1.2 Variantes et classification de Lawler . . . 45
5.1.3 Applications . . . 45
5.2 Résultats de complexité . . . 45
5.2.1 Modèle en parallèle . . . 46
5.2.2 Modèle en série . . . 46
5.2.3 Conclusion . . . 46
5.3 État de l'art . . . 47
5.3.1 Jeu d'instances . . . 47
5.3.2 Méthodes de résolution . . . 47
5.4 Orientation de nos travaux . . . 48

II Contribution

6 Ordonnancement d'atelier au plus tôt 51
6.1 Modèle en ordonnancement sous contraintes . . . 52
6.1.1 Contraintes disjonctives . . . 52
6.1.2 Contraintes temporelles . . . 52
6.1.3 Contraintes supplémentaires . . . 53
6.1.4 Stratégie de branchement . . . 54
6.2 Algorithme de résolution . . . 55
6.2.1 Principe de l'algorithme . . . 55
6.2.2 Solution initiale . . . 57
6.2.3 Techniques de redémarrage . . . 57
6.3 Évaluations expérimentales . . . 58
6.3.1 Réglage des paramètres . . . 59
6.3.2 Analyse de sensibilité . . . 60
6.3.3 Comparaison avec d'autres approches . . . 64
6.4 Conclusion . . . 67

7 Heuristiques d'arbitrage dans un atelier **69**
7.1 Modèles en ordonnancement sous contraintes . . . 70
7.1.1 Contraintes temporelles . . . 70
7.1.2 Stratégie de branchement . . . 70
7.1.3 Notes sur les modèles . . . 71
7.2 Algorithme de résolution . . . 72
7.3 Évaluations expérimentales . . . 72
7.3.1 Problèmes d'open-shop . . . 72
7.3.2 Problèmes de job-shop . . . 74
7.4 Conclusion . . . 75

8 Minimisation du retard algébrique maximal sur une machine à traitement par fournées **77**
8.1 Modèle en programmation par contraintes . . . 78
8.2 Description de la contrainte globale `sequenceEDD` . . . 79
8.2.1 Relaxation du problème . . . 80
8.2.2 Filtrage de la variable objectif . . . 81
8.2.3 Filtrage basé sur le coût du placement d'une tâche . . . 82
8.2.4 Filtrage basé sur le coût du nombre de fournées non vides . . . 83
8.3 Stratégie de branchement . . . 84
8.4 Expérimentations . . . 85
8.4.1 Performance des règles de filtrage . . . 85
8.4.2 Performance des heuristiques de sélection de valeur . . . 87
8.4.3 Comparaison avec un modèle en programmation mathématique . . . 88
8.4.4 Comparaison avec une approche par branch-and-price . . . 89
8.5 Conclusion . . . 89
Annexe 8.A : un algorithme quadratique pour le filtrage du placement d'une tâche . . . 90

9 Implémentation dans le solveur de contraintes `choco` **93**
9.1 Le modèle . . . 94
9.1.1 Variables . . . 94
9.1.2 Contraintes . . . 95
9.2 Le solveur . . . 96
9.2.1 Lecture du modèle . . . 96
9.2.2 Stratégie de branchement . . . 97
9.2.3 Redémarrages . . . 98
9.2.4 Résolution d'un modèle . . . 99
9.3 Contraintes de choco . . . 99
9.3.1 cumulative . . . 100
9.3.2 disjunctive . . . 101
9.3.3 useResources (en développement) . . . 102
9.3.4 precedence . . . 103
9.3.5 precedenceDisjoint . . . 103
9.3.6 precedenceReified . . . 103
9.3.7 precedenceImplied . . . 104
9.3.8 disjoint (décomposition) . . . 104
9.3.9 pack . . . 104
9.3.10 Reformulation de contraintes temporelles . . . 106
9.4 Construction du modèle disjonctif . . . 106
9.4.1 Traitement des contraintes temporelles . . . 107
9.4.2 Traitement des contraintes de partage de ressource . . . 107
9.4.3 Déduction de contraintes « coupe-cycle » . . . 109
9.5 CSP Solver Competition 2009 . . . 110
9.5.1 Catégorie 2-ARY-INT . . . 110
9.5.2 Catégorie Alldiff+Cumul+Elt+WSum . . . 113

9.6 Conclusion 113

10 Conclusion **115**
10.1 Problèmes d'atelier 116
10.2 Problème de fournées 116
10.3 Solveur choco 116

Annexes

A Outils `choco` : développement et expérimentation **119**
A.1 Procédure générique de traitement d'une instance 119
A.2 Assistant à la création d'un programme en ligne de commande 121
A.3 Intégration dans une base de données 121

B Tutoriel `choco` : ordonnancement et placement **123**
B.1 Gestion de projet : construction d'une maison 123
B.1.1 Approche déterministe 124
B.1.2 Approche probabiliste 126
B.2 Ordonnancement cumulatif 129
B.3 Placement à une dimension 132
B.3.1 Traitement d'une instance 132
B.3.2 Création d'une commande 134
B.3.3 Cas d'utilisation 135

C Problème de collectes et livraisons avec contraintes de chargement bidimensionnelles **139**

D Problème d'allocation de ressources et d'ordonnancement pour la maintenance **149**

Bibliographie **165**

Résumés **178**

Table des figures

2.1 Exemple de CSP : n reines. . . . 12
2.2 Exemples d'affectation : 4 reines. . . . 12
2.3 Algorithme *backtrack* : 4 reines. . . . 14
2.4 Exemple de CSP où l'arc-consistance est incomplète. . . . 15
2.5 Exemple de CSP utilisant des contraintes globales: n reines. . . . 15
2.6 Algorithmes de recherche rétrospectifs : 8 reines. . . . 16
2.7 Influence des heuristiques de sélection sur l'algorithme *backtrack* : 4 reines. . . . 18
2.8 Ordre d'exploration des branches par différentes recherches arborescentes. . . . 19
2.9 Description de la procédure d'optimisation bottom-up. . . . 20
2.10 Description de la procédure d'optimisation top-down. . . . 21
2.11 Description de la procédure d'optimisation dichotomic-bottom-up. . . . 22
2.12 Description de la procédure d'optimisation destructive-upper-bound. . . . 23

3.1 Notations relatives à l'exécution d'une tâche. . . . 26
3.2 Hiérarchie des types de ressources. . . . 29
3.3 Construction et d'arbitrage d'un graphe disjonctif. . . . 32
3.4 Une condition nécessaire mais pas suffisante pour le problème de faisabilité. . . . 34

4.1 Un ordonnancement optimal de délai total 1170 pour l'instance d'open-shop `GP04-01`. . . . 36

5.1 Un ordonnancement optimal de retard maximal -90 pour l'instance `bp10-07`. . . . 44

6.1 Exemples de stratégie de branchement. . . . 55
6.2 Structure de l'algorithme `RRCP`. . . . 56
6.3 Écart à l'optimum de la solution initiale et influence de `CROSH` sur les performances de `RRCP`. 61
6.4 Influence des redémarrages et de l'enregistrement des *nogoods*. . . . 62
6.5 Influence des redémarrages avec enregistrement des *nogoods* et équivalence des stratégies de redémarrage. . . . 63

7.1 Comparaison des modèles *Light* et *Heavy* sur le problème d'open-shop. . . . 73

8.1 Deux exemples de construction de la relaxation $I(\mathcal{A})$. . . . 81
8.2 Exemple de calcul de $L(\mathcal{A}_3 \cap \{M \leftarrow 4\})$ pour une affectation (vide) $\mathcal{A}_3$. . . . 84
8.3 Comparaison des règles de filtrage en utilisant *complete decreasing first fit*. . . . 86
8.4 Comparaison des heuristiques de sélection de valeur. . . . 87
8.5 Comparaison à la formulation mathématique. . . . 89
8.6 Anticipation de l'effet du placement d'une tâche sur la séquence de blocs. . . . 90
8.7 Solution de la relaxation après le placement de la tâche 4. . . . 92

9.1 Élimination des arbitrages créant un cycle de longueur 3 dans le graphe disjonctif. . . . 109
9.2 Résolution de 100 problèmes d'open-shop (CSC'2009). . . . 111
9.3 Résolution de 126 problèmes de job-shop (CSC'2009) 111
9.4 Nombre d'instances résolues dans une limite de temps (2-ARY-INT). . . . 112
9.5 Nombre d'instances résolues dans une limite de temps (Alldiff+Cumul+Elt+WSum). . . . 113

B.1 Visualisation des graphes disjonctifs (activée à l'étape 5) pendant les étapes 6 et 7. . . . 127

B.2 Probabilité que le projet finisse au plus tard à une date D. 129
B.3 Une instance de notre problème d'ordonnancement cumulatif. 129
B.4 Une solution optimale de l'instance décrite en figure B.3. 132
B.5 Une solution optimale de l'instance N1C3W1_A. 136
B.6 Aperçu d'un rapport généré par oobase. 137

Liste des tableaux

5.1 Complexité des problèmes de fournées sans contrainte de précédence ni date de disponibilité. 47

6.1 Identification de bons paramètres pour les stratégies de redémarrage. 60
6.2 Résultats sur les trois instances de Brucker les plus difficiles. 63
6.3 Résultats sur les instances de Taillard. 66
6.4 Résultats sur les instances de Brucker. 67
6.5 Résultats sur les instances de Guéret-Prins. 68

7.1 Temps de résolution des 11 instances les plus difficiles du problème d'open-shop. 73
7.2 Comparaison des heuristiques de sélection de disjonction sur le problème d'open-shop. . . 74
7.3 Résultats sur le problème de job-shop. 75

8.1 Qualité des bornes inférieures destructives. 86

9.1 Méthodes pour la création d'une ou plusieurs tâches (`TaskVariable`). 95
9.2 Aperçu des méthodes publiques de la classe `TaskVar`. 97
9.3 Méthodes de résolution du `Solver`. 99
9.4 Principales méthodes de la classe `PackModel`. 106
9.5 Reformulation des contraintes temporelles. 106

B.1 Estimation probabiliste des durées des tâches. 128
B.2 Longueurs et variances des chemins critiques. 128

Listings

1 Calculer toutes les solutions du problème des n reines avec choco. 24
2 Méthode principale de la classe `AbstractInstanceModel` 119

Liste des Algorithmes

- Procédure bottom-up(lb,ub) 20
- Procédure top-down(lb,ub) 21
- Procédure dichotomic-bottom-up(lb,ub) 22
- Procédure destructive-lower-bound(lb,ub) 22
- Procédure destructive-upper-bound(lb,ub) 23

- Procédure jobCBF 91

- Procédure buildDisjModT 108
- Procédure buildDisjModR 108
- Procédure postCoupeCycle 109

Chapitre 1

Introduction

L'organisation et la gestion de la production conditionnent le succès des projets du monde de l'entreprise et de la recherche. Dans ce processus, la fonction ordonnancement vise à organiser l'utilisation des ressources technologiques ou humaines pour répondre à une demande ou satisfaire un plan de production préparé par la fonction planification. Ainsi, des programmes ambitieux privés ou publics ont recours à la fonction ordonnancement pour appréhender la complexité, améliorer les délais ou même s'adapter à des évènements imprévus. Les problèmes d'ordonnancement apparaissent dans de nombreux domaines : l'industrie (atelier, gestion de production), la construction (suivi de projets), mais aussi l'informatique (gestion des processus) et l'administration (emplois du temps).

Un problème d'ordonnancement est composé de façon générale d'un ensemble de tâches soumises à certaines contraintes, et dont l'exécution nécessite des ressources. Résoudre un problème d'ordonnancement consiste à organiser ces tâches, c'est-à-dire à déterminer leurs dates de démarrage et d'achèvement, et à leur attribuer des ressources, de telle sorte que les contraintes soient respectées. Les problèmes d'ordonnancement sont très variés. Ils sont caractérisés par un grand nombre de paramètres relatifs aux tâches (morcelables ou non, indépendantes ou non, durées fixes ou non), aux ressources (renouvelables ou consommables), aux types de contraintes portant sur les tâches (contraintes temporelles, fenêtres de temps ...), aux critères d'optimalité liés au temps (délai total, délai moyen, retards ...), aux ressources (quantité utilisée, taux d'occupation ...) ou à d'autres coûts (production, lancement, stockage ...). Parmi tous ces problèmes, nous nous intéresserons à l'optimisation de critères réguliers [1] pour des problèmes d'atelier et de fournées classés NP-Difficiles.

Dans un problème d'atelier, une pièce doit être usinée ou assemblée sur différentes machines. Chaque machine est une ressource disjonctive, c'est-à-dire qu'elle ne peut exécuter qu'une tâche à la fois, et les tâches sont liées exclusivement par des contraintes d'enchaînement. Plus précisément, les tâches sont regroupées en n entités appelées travaux ou lots. Chaque lot est constitué de m tâches à exécuter sur m machines distinctes. Il existe trois types de problèmes d'atelier, selon la nature des contraintes liant les tâches d'un même lot. Lorsque l'ordre de passage de chaque lot est fixé et commun à tous les lots, on parle d'atelier à cheminement unique (*flow-shop*). Si cet ordre est fixé mais propre à chaque lot, il s'agit d'un atelier à cheminements multiples (*job-shop*). Enfin, si le séquencement des tâches des travaux n'est pas imposé, on parle d'atelier à cheminements libres (*open-shop*). Un critère d'optimalité souvent étudié est la minimisation du délai total de l'ordonnancement (*makespan*). Ce critère est particulièrement intéressant puisque les ordonnancements au plus tôt constituent des sous-ensembles dominants [2] pour de nombreux critères réguliers. De plus, l'analyse des ordonnancements au plus tôt permet de déterminer des chemins critiques, c'est-à-dire des chemins sur lesquels tout retard a des conséquences sur toute la chaîne, ou des goulots d'étranglement, c'est-à-dire les étapes qui vont limiter la production de tout l'atelier.

Une machine à traitement par fournées permet de traiter plusieurs tâches en une seule opération, une fournée. Les dates de début et de fin des tâches appartenant à une même fournée sont identiques. Les machines à traitement par fournées diffèrent généralement par les contraintes sur l'ordonnancement des

1. Un critère d'évaluation numérique ou fonction objectif est dit régulier si l'on ne peut le dégrader en avançant l'exécution d'une tâche.
2. Un ensemble de solutions d'un problème d'optimisation est dit dominant s'il contient au moins une solution optimale.

fournées ou celles restreignant les fournées réalisables. La résolution exacte des problèmes de fournées est encore un sujet récent et peu abordé. Nous nous intéresserons à la minimisation du retard algébrique maximal de n tâches de différentes tailles sur une machine à traitement par fournées pour laquelle la somme des tailles des tâches d'une fournée réalisable ne doit pas excéder la capacité b de la machine.

Durant les dernières décennies, la Programmation Par Contraintes (PPC) est devenue une approche efficace pour modéliser et résoudre les problèmes d'ordonnancement. La PPC permet de séparer le modèle décrit grâce à un language déclaratif (tâches, ressources, contraintes, objectif) des algorithmes employés durant la résolution. Ces travaux visent à faciliter l'interaction entre les modèles et les décideurs par l'intégration d'outils utiles dans un contexte d'aide à la décision. Ainsi, la flexibilité de modélisation, les méthodes structurelles de reformulation, la gestion dynamique des contraintes permettent d'appliquer des méthodes de simulation, de planification, ou même de diagnostic. Les méthodes de résolution telles que la réduction de domaines et la propagation de contraintes combinées à des algorithmes de recherche efficaces, exhaustifs ou non, ont permis la résolution de nombreux problèmes industriels. De nos jours, ces méthodes sont de plus en plus couplées avec des techniques de Recherche Opérationnelle (RO), telles que la programmation linéaire, entière ou mixte, pour élaborer des algorithmes efficaces dédiés à l'optimisation.

1.1 Objectifs

L'objectif de cette thèse est d'étudier la résolution exacte des problèmes d'atelier et de fournées grâce à la programmation par contraintes. En effet, l'étude et la résolution efficace de ces problèmes permettent de mieux appréhender des systèmes industriels complexes dans lesquels ils sont des composantes essentielles. Par exemple, un problème d'atelier est une composante d'un problème d'ordonnancement cumulatif ou de fournées. De la même manière, les problèmes à une machine à traitement par fournées sont des composantes des problèmes à plusieurs machines.

Dans un premier temps, nous proposerons une *approche flexible et efficace pour la résolution exacte des problèmes d'atelier*. Sa conception reposera sur la classification et l'étude systématique (empirique et expérimentale) de différents modèles et stratégies de recherche. Cette étude identifiera un nombre restreint de configurations dominantes, mais sans fournir pour autant un critère permettant le choix a priori de la meilleure configuration. Nous proposerons aussi plusieurs nouvelles heuristiques de sélection de variable inspirées d'heuristiques dynamiques basées sur les domaines et les degrés [3]. Enfin, nous insisterons sur la méthodologie employée pour l'étude des interactions entre les modèles et les stratégies de recherche qui sont souvent négligées par les utilisateurs non spécialistes.

Dans un second temps, nous proposerons une *nouvelle approche pour la résolution exacte d'un problème de fournées*. problème maître de la décomposition est un problème de placement pour la construction des fournées réalisables et le sous-problème consiste à ordonnancer les fournées. La résolution sera renforcée par des techniques de filtrage basées sur les coûts exploitant une relaxation du problème définissant une nouvelle borne inférieure sur le retard algébrique maximal. Ces techniques, classiques en recherche opérationnelle, permettent de réduire l'espace de recherche et de mieux gérer les informations relatives au critère d'optimalité. Un modèle en programmation entière faiblement basé sur la décomposition sera aussi étudié à titre de comparaison. Nous montrerons la validité de notre décomposition pour d'autres problèmes où l'ordonnancement optimal des fournées respecte certaines conditions.

Pour finir, nous discuterons de la *diffusion des techniques d'ordonnancement sous contraintes par le biais de leur intégration dans le solveur de contraintes* `choco`. Nous nous intéresserons donc à la conception, à l'implémentation et à l'utilisation des modules d'ordonnancement et de placement développés au cours de cette thèse. La clarté, la flexibilité et la simplicité des algorithmes et structures de données seront discutées au regard de leur complexité spatiale et temporelle. Au cours de ce travail, les échanges avec les autres chercheurs et utilisateurs ont permis d'améliorer et de corriger les spécifications et quelquefois l'implémentation, mais surtout d'élargir considérablement nos axes de réflexion. Ainsi, nous aborderons une extension importante du problème d'ordonnancement, l'allocation de ressources, c'est-à-dire la prise en compte du choix des ressources utilisées pour réaliser les tâches et de l'ordonnancement des tâches sur les ressources utilisées.

3. Le degré d'une variable est le nombre de contraintes auxquelles elle appartient.

1.2 Contribution

Ce document décrit notre apport dans le domaine de l'ordonnancement sous contraintes selon trois axes principaux :

Méthodologique : nous proposons une approche méthodologique, adaptée aux problèmes d'ordonnancement d'atelier et de fournées, pour la modélisation et la sélection des algorithmes de résolution.

Technique : nous introduisons de nouvelles techniques (heuristiques de recherche, contraintes globales) pour combler certaines lacunes identifiées grâce à notre méthodologie.

Logiciel : l'implémentation de ces techniques dans un solveur de contraintes facilite leur diffusion et leur mise en œuvre dans diverses applications.

Nous établissons une classification des modèles et algorithmes de résolution pour les problèmes d'atelier. Nous distinguons les modèles en fonction de la décomposition des contraintes globales, l'algorithme de recherche (prospectif, rétrospectif, redémarrages), des variables de décision et des heuristiques de sélection de variable et de valeur.
Nous proposons ensuite une nouvelle famille d'heuristiques dynamiques de sélection de variable utilisant activement la décomposition des contraintes globales. Ces heuristiques exploitent un mécanisme d'apprentissage du degré des variables pour guider la recherche à travers les parties difficiles ou inconsistantes. Les évaluations révéleront l'intérêt de la décomposition des contraintes globales et remettront en question l'utilisation systématique de celles-ci dans le contexte des problèmes d'atelier. De plus, nous identifierons certaines composantes critiques de notre approche comme la qualité de la solution initiale et l'utilisation de techniques de redémarrage.
Un module d'ordonnancement pour le solveur de contraintes `choco` permet de reproduire nos expérimentations, mais aussi de définir de nouvelles contraintes et stratégies de recherche en spécialisant des classes génériques basées sur un type de variable modélisant une tâche.

Par ailleurs, l'intérêt des chercheurs en programmation par contraintes pour les problèmes de fournées est encore récent. Nous proposons d'abord un modèle par décomposition exploitant la richesse du langage déclaratif de modélisation. Le problème maître de la décomposition est un problème de placement pour la construction des fournées réalisables et le sous-problème consiste à ordonnancer les fournées. La modélisation du problème maître utilise les résultats des travaux en placement sous contraintes, notamment des contraintes globales et des stratégies de recherche. La nature du sous-problème change selon le critère d'optimalité.
Nous montrerons aussi l'intérêt d'une hybridation avec des techniques classiques en recherche opérationnelle pour la minimisation du retard algébrique maximal des fournées. Ainsi, nous proposons une nouvelle contrainte globale basée sur une relaxation du problème qui réalise un filtrage basé sur les coûts. Les évaluations montreront que ce filtrage basé sur les coûts entraîne une amélioration significative des performances. L'impact des décisions sur le critère d'optimisation est souvent géré moins efficacement par un solveur de contraintes que par un solveur de programmation mathématique. Finalement, nous proposerons une nouvelle heuristique de sélection de variable où l'allocation des tâches aux fournées est basée sur l'ordonnancement des fournées plutôt que sur leur charge.

Nos contributions sur le solveur de contraintes Choco concernent principalement l'ordonnancement, le placement et les stratégies de recherche, mais touchent aussi les interfaces de modélisation, débogage, expérimentation et de visualisation. Face à la quasi-impossibilité de proposer une liste exhaustive de contraintes, consistances et algorithmes/stratégies de résolution, nous chercherons une flexibilité accrue grâce à un langage déclaratif de haut niveau et des interfaces pour la spécification de contraintes et d'algorithmes de résolution. Au début de ce travail, seule une contrainte cumulative définie sur des variables entières était disponible. L'introduction d'un nouveau type de variable représentant les tâches offre dorénavant un accès à un catalogue de contraintes et de stratégies dédiées à l'ordonnancement. De plus, l'utilisateur peut définir de nouvelles contraintes et stratégies dédiées à l'ordonnancement qui héritent de classes abstraites ou génériques. Parallèlement, un module de placement en une dimension, développé dans le cadre des problèmes de fournées, met à disposition une contrainte globale ainsi que quelques stratégies de recherche et d'élimination des symétries.
L'implémentation des stratégies de recherche doit être robuste et efficace pour affronter la variété des applications possibles tout en restant flexible pour permettre leur modification. Nous discuterons aussi de l'implémentation des procédures d'optimisation et des redémarrages. Quelques outils génériques de

traitement d'une instance et d'expérimentation seront présentés pour leur intérêt pratique. En effet, ils réduisent l'effort d'implémentation, facilitent le diagnostic, améliorent la communication entre utilisateurs et procurent au final un gain de temps appréciable.

1.3 Organisation du document

L'objet de cette thèse est de présenter nos contributions dans le domaine de l'ordonnancement par contraintes par rapport aux travaux passés et présents. Nous avons choisi de privilégier la modélisation et la résolution complète de problèmes d'atelier et de fournées rencontrés en pratique plutôt qu'un aspect spécifique de l'ordonnancement sous contraintes. Ceci nous a conduit à l'organisation du manuscrit en deux parties décrites ci-dessous.

La première partie décrit le contexte de notre étude. Une présentation synthétique des concepts liés à la programmation par contraintes, l'optimisation sous contraintes et à l'ordonnancement sous contraintes est proposée aux chapitres 2 et 3. Les deux chapitres suivants introduisent deux familles de problèmes d'ordonnancement en résumant les principaux résultats de complexité et l'état de l'art. Le chapitre 4 présente un problème général d'ordonnancement d'atelier puis trois problèmes particuliers : l'open-shop, le flow-shop et le job-shop. Le chapitre 5 présente les problèmes de fournées en insistant sur ceux où les tâches ont des tailles différentes.

La seconde partie de ce document présente les éléments de contribution de cette thèse. Dans un premier temps, nous décrivons notre approche pour la résolution de problèmes d'atelier. Le chapitre 6 décrit les principes de notre algorithme de résolution basé sur l'amélioration d'une solution initiale obtenue de manière heuristique par une recherche arborescente avec redémarrages. Le chapitre 7 décrit plusieurs variantes de notre algorithme avec de nouvelles heuristiques de sélection de variable. Ensuite, le chapitre 8 décrit notre modèle basé sur une décomposition utilisant une contrainte globale réalisant un filtrage basé sur les coûts pour la résolution d'un problème de fournées où les tâches ont des tailles différentes. Finalement, le chapitre 9 aborde la conception et l'implémentation de plusieurs modules du solveur de contraintes `choco`. Finalement, nous concluons ce manuscrit en réalisant un bilan de cette thèse et en discutant des perspectives.

L'annexe A introduit des outils de développement et d'expérimentation disponibles dans le solveur `choco`. L'annexe B est un tutoriel `choco` pour l'ordonnancement et le placement sous contraintes. L'annexe C est une étude préliminaire sur la résolution de problèmes de tournées de véhicules. L'annexe D décrit des travaux préliminaires sur un problème d'allocation de ressources.

1.4 Diffusion scientifique

Les différents travaux présentés dans ce document ont fait l'objet de diverses publications et communications. Au début de cette thèse, une étude préliminaire sur les problèmes de tournées de véhicules avec contraintes de chargement nous a convaincu que le solveur `choco` n'était pas encore mûr pour traiter de tels problèmes malgré des travaux réalisés au sein de l'équipe sur des sujets connexes. Cette étude a fait l'objet d'un rapport technique [1] et d'une communication [2] lors d'un atelier de la conférence internationale *CPAIOR 2008* (*International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming*). Par contre, cette étude a révélé que les techniques d'ordonnancement sous contraintes jouaient un rôle crucial dans la résolution des problèmes de placement, mais aussi dans les problèmes de tournées avec fenêtres de temps.
Notre attention s'est alors portée sur les problèmes d'ordonnancement disjonctif qui sont souvent des composantes essentielles dans les problèmes d'ordonnancement ou de placement. Dans ce contexte, la méthode de résolution pour les problèmes d'atelier a fait l'objet d'un rapport technique [3], d'une communication lors des *Journées de l'Optimisation 2009* à Montréal, et finalement d'une publication [4] dans *Journal of Computing*. Entre-temps, des travaux complémentaires [5] proposant de nouvelles heuristiques de sélection ont quant à eux fait l'objet d'une publication dans la conférence internationale *CP 2009* (*International Conference on Principles and Practice of Constraint Programming*). En parallèle, des travaux préliminaires sur les problèmes d'allocation de ressources ont fait l'objet d'une communication [6] lors d'un atelier de la conférence internationale *CP 2010* et d'un rapport technique [7].

Finalement, les résultats préliminaires sur les problèmes de fournées ont fait l'objet d'une première communication lors des *Journées de l'optimisation 2010*, puis d'un rapport technique [8]. La version définitive [9] a été soumise pour publication dans *European Journal of Operational Research* et a fait l'objet d'une communication sous la forme d'un résumé étendu lors de la conférence internationale *CPAIOR 2011*.

Bibliographie

[1] Arnaud Malapert, Christelle Guéret, Narendra Jussien, André Langevin et Louis-Martin Rousseau : Two-dimensional pickup and delivery routing problem with loading constraints. Rapport technique CIRRELT-2008-37, Centre Inter-universitaire de Recherche sur les Réseaux d'Entreprise, la Logistique et le Transport, Montréal, Canada, 2008.

[2] Arnaud Malapert, Christelle Guéret, Narendra Jussien, André Langevin et Louis-Martin Rousseau : Two-dimensional pickup and delivery routing problem with loading constraints. *In First CPAIOR Workshop on Bin Packing and Placement Constraints (BPPC'08)*, Paris, France, mai 2008.

[3] Arnaud Malapert, Hadrien Cambazard, Christelle Guéret, Narendra Jussien, André Langevin et Louis-Martin Rousseau : An optimal constraint programming approach to solve the open-shop problem. Rapport technique CIRRELT-2009-25, Centre Inter-universitaire de Recherche sur les Réseaux d'Entreprise, la Logistique et le Transport, Montréal, Canada, 2009.

[4] Arnaud Malapert, Hadrien Cambazard, Christelle Guéret, Narendra Jussien, André Langevin et Louis-Martin Rousseau : An optimal constraint programming approach to solve the open-shop problem. *Journal of Computing*, in press, accepted manuscript, 2011.

[5] Diarmuid Grimes, Emmanuel Hebrard et Arnaud Malapert : Closing the open shop : Contradicting conventional wisdom. *In Principles and Practice of Constraint Programming - CP 2009*, volume 5732 de *Lecture Notes in Computer Science*, pages 400–408. Springer Berlin / Heidelberg, 2009.

[6] Aliaa M. Badr, Arnaud Malapert et Kenneth N. Brown : Modelling a maintenance scheduling problem with alternative resources. *In The 9th International Workshop on Constraint Modelling and Reformulation (CP10)*, St. Andrews, Scotland, september 2010.

[7] Aliaa M. Badr, Arnaud Malapert et Kenneth N. Brown : Modelling a maintenance scheduling problem with alternative resources. Rapport technique 10/3/INFO, École des Mines de Nantes, Nantes, France, 2010.

[8] Arnaud Malapert, Christelle Guéret et Louis-Martin Rousseau : A constraint programming approach for a batch processing problem with non-identical job sizes. Rapport technique 11/6/AUTO, École des Mines de Nantes, Nantes, France, june 2011.

[9] Arnaud Malapert, Christelle Guéret et Louis-Martin Rousseau : A constraint programming approach for a batch processing problem with non-identical job sizes. *European Journal of Operational Research*, in revision, submitted manuscript, april 2011.

Première partie

Contexte de l'étude

Chapitre 2

Programmation par contraintes

Nous présentons brièvement les principes de la programmation par contraintes pour la résolution de problèmes de satisfaction de contraintes et d'optimisation sous contraintes. Nous insisterons sur le rôle crucial joué par les stratégies de recherche notamment par le biais des heuristiques de sélection de variable et de valeur. Nous rappellerons aussi le principe des procédures d'optimisation qui transforme un problème d'optimisation sous contraintes en une série de problèmes de satisfaction de contraintes. Avant de conclure, nous passerons en revue les principaux solveurs rencontrés lors de nos recherches en précisant leurs fonctionnalités propres à l'ordonnancement.

Sommaire

2.1 Modélisation d'un problème par des contraintes . . . 10
2.1.1 Problème de satisfaction de contraintes . . . 11
2.1.2 Problème d'optimisation sous contraintes . . . 12
2.1.3 Exemple : le problème des n reines . . . 12
2.2 Méthodes de résolution . . . 13
2.2.1 Algorithmes simples de recherche . . . 13
2.2.2 Filtrage et propagation des contraintes . . . 13
2.2.3 Contraintes globales . . . 15
2.2.4 Algorithmes avancés de recherche . . . 15
2.3 Stratégies de recherche . . . 16
2.3.1 Méthodes de séparation . . . 16
2.3.2 Heuristiques de sélection . . . 17
2.4 Procédures d'optimisation . . . 19
2.4.1 Procédure bottom-up . . . 20
2.4.2 Procédure top-down . . . 20
2.4.3 Procédure dichotomic-bottom-up . . . 21
2.4.4 Procédures incomplètes . . . 21
2.5 Solveurs de contraintes . . . 23
2.6 Conclusion . . . 24

La programmation par contraintes est une technique de résolution des problèmes combinatoires complexes issue de la programmation logique et de l'intelligence artificielle et apparue à la fin des années 1980. Elle consiste à modéliser un problème par un ensemble de relations logiques, des contraintes, imposant des conditions sur l'instanciation possible d'un ensemble de variables définissant une solution du problème. Un solveur de contraintes calcule une solution en instanciant chacune des variables à une valeur satisfaisant simultanément toutes les contraintes. De nos jours, de nombreuses techniques issues de la recherche opérationnelle, de la programmation mathématique ou même de la recherche locale sont appliquées grâce à la séparation entre un langage de modélisation déclaratif et les algorithmes employés durant la résolution. Les types d'application traités par la programmation par contraintes sont très variés. Parmi les domaines d'application les plus classiques, on pourra citer la gestion du temps ou d'affectation de

ressources, la planification et l'ordonnancement, mais aussi la simulation comportementale des systèmes, la vérification des spécifications ou le diagnostic des pannes. La mise en œuvre des contraintes dans un contexte industriel peut se faire sous diverses formes : certaines entreprises utilisent des solveurs commerciaux, d'autres font seulement appel occasionnellement à des bibliothèques de gestion de contraintes, d'autres enfin développent des prototypes servant uniquement à la modélisation du problème qui peuvent ensuite être recodés de manière plus efficace dans un langage de programmation traditionnel ou à l'aide d'un module numérique. En revanche, la manière de poser le problème peut modifier considérablement les performances du programme. De plus, l'évaluation du temps d'exécution peut être si subtile et dépendante du problème, qu'il n'est pas possible d'énoncer des règles claires. Ces aspects peuvent entraîner une frustration au regard des espérances qu'elle fait naître auprès des utilisateurs. Ainsi, il est critique d'évaluer la taille d'un problème et la combinatoire qu'il engendre avant d'espérer pouvoir le résoudre.

Les problèmes d'ordonnancement et de placement forment une classe de problèmes d'optimisation combinatoire généralement complexe. Cependant, la classification de ces problèmes et l'étude de leur complexité ont permis d'exhiber de nombreux problèmes polynomiaux. Un large éventail de méthodes complexes d'optimisation approchées ou exactes ont été appliquées sur les problèmes non polynomiaux avec des succès divers. La programmation par contraintes a connu quelques-uns de ses plus retentissants succès lors de la résolution de problèmes d'ordonnancement. En effet, la combinaison des méthodes de recherche opérationnelle (théorie des graphes, programmation mathématique, algorithmes de permutation), avec des modèles de représentation et de traitement des contraintes de l'intelligence artificielle (satisfaction de contraintes, propagation de contraintes, langages déclaratifs de modélisation) ont permis de mettre au point des outils facilitant l'interaction entre les modèles et les décideurs en ajoutant des méthodes d'analyse (vérification de la consistance, caractérisation de l'espace des solutions) à des algorithmes de résolution efficaces (recherche arborescente, recherche locale, optimisation). La conception de mécanismes efficaces et généraux de propagation de contraintes qui réduisent l'espace de recherche sera discutée au chapitre suivant dans le cas des contraintes temporelles et de partage de ressource utilisées dans les problèmes traités dans cette thèse.

La propagation des contraintes seule est généralement insuffisante pour exhiber une solution, il est alors nécessaire d'appliquer un algorithme de recherche pour trouver une ou plusieurs solutions. La méthode la plus commune est la recherche arborescente avec retour arrière, mais de nombreuses variantes existent incluant des algorithmes prospectifs, rétrospectifs ou mixtes. D'autres algorithmes de recherche s'inspirent de la recherche locale en adaptant les notions de voisinage ou de population. Il est donc nécessaire d'identifier les techniques adaptées à la résolution d'un type de problème donné. Les heuristiques de sélection déterminent l'ordre de traitement des variables et des valeurs dans une recherche arborescente et, de ce fait, influencent ses performances.

Nous discutons dans ce chapitre des principes fondateurs de la PPC. Le lecteur est invité à se référer à Rossi *et al.* [10] pour un état de l'art approfondi. Dans un premier temps, nous décrivons le modèle de déclaration des problèmes de satisfaction de contraintes et d'optimisation sous contraintes (section 2.1) avant de discuter des méthodes de résolution des problèmes à base de contraintes et des techniques permettant d'accélérer le processus de résolution (section 2.2). Ensuite, nous abordons quelques notions relatives aux stratégies de recherche (section 2.3) et à l'optimisation sous contraintes (section 2.4). Avant de conclure, nous présentons les principes de quelques solveurs de contraintes qui mettent à disposition des utilisateurs une bibliothèque pour la modélisation et la résolution de problèmes variés (section 2.5).

2.1 Modélisation d'un problème par des contraintes

Nous définissons dans cette section les différents éléments permettant de modéliser un problème combinatoire par une conjonction de contraintes. Dans une première partie, nous définissons les problèmes de satisfaction de contraintes puis nous présentons les problèmes d'optimisation sous contraintes dont les solutions maximisent ou minimisent la valeur d'une fonction de coût.

2.1.1 Problème de satisfaction de contraintes

Une contrainte est une relation logique établie entre différentes variables, chacune prenant sa valeur dans un ensemble qui lui est propre, appelé domaine. Une contrainte sur un ensemble de variables restreint les valeurs que peuvent prendre simultanément ses variables. Une contrainte est déclarative et relationnelle puisqu'elle définit une relation entre les variables sans spécifier de procédure opérationnelle pour assurer cette relation. L'arité d'une contrainte est égale à la cardinalité de son ensemble de variables, appelé son scope. Par exemple, la contrainte binaire $(x, y) \in \{(0, 1), (1, 2)\}$ précise que la variable x ne peut prendre la valeur 0 que si la variable y prend la valeur 1 et que la variable y peut prendre la valeur 2 uniquement si x prend la valeur 1. Une telle contrainte est définie en extension, c'est-à-dire en spécifiant explicitement les tuples de valeurs qu'elle autorise (ou interdit). Une contrainte peut également être définie en intension par une équation mathématique ($x+y \leq 4$), une relation logique simple ($x \leq 1 \Rightarrow y \neq 2$) ou même par un prédicat portant sur n variables ($\texttt{allDifferent}(x_1, \ldots, x_n)$). Une telle contrainte est appelée contrainte globale. Nous reviendrons sur la définition et l'intérêt des contraintes globales tout au long de cette thèse. La définition d'une nouvelle contrainte est aisée puisque la seule opération indispensable est le test de consistance qui détermine si une instanciation de ces variables satisfait la contrainte.

Un problème de satisfaction de contraintes (CSP) est défini formellement par un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ représentant respectivement un ensemble fini de variables $\mathcal{X}$, une fonction $\mathcal{D}$ affectant à chaque variable $x \in \mathcal{X}$ son domaine $\mathcal{D}(x)$ et un ensemble fini de contraintes $\mathcal{C}$. Le domaine initial d'une variable $\mathcal{D}_0(x)$ représente l'ensemble de valeurs auxquelles la variable x peut être instanciée avant le début de la résolution. Chaque contrainte $c \in \mathcal{C}$ est une relation multidirectionnelle portant sur un sous-ensemble de variables noté $var(c) \subseteq \mathcal{X}$ restreignant les valeurs que ces variables peuvent prendre simultanément. Les problèmes traités dans cette thèse sont des CSP discrets ($\mathcal{D}(x) \in \mathbb{Z}$) et statiques, c'est-à-dire l'ensemble des contraintes ne change pas au cours de la résolution.
Le graphe de contraintes d'un CSP binaire est un graphe simple où les sommets représentent les variables et les arêtes représentent les contraintes. Il y a une arête entre les sommets représentant les variables x et y si et seulement s'il existe une contrainte c telle que $var(c) = \{x, y\}$. 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 hyperarêtes qui relient des sous-ensembles de variables impliquées dans une même contrainte. Dans ce cas, on a un hypergraphe 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 auxquels 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. De même, on peut définir les graphes de consistance/inconsistance dont les sommets sont les valeurs et les hyperarcs sont les tuples autorisés/interdits.
Une instanciation $x \leftarrow v$ consiste à affecter à une variable x une valeur v appartenant à son domaine $\mathcal{D}(x)$. À chaque étape de la résolution, une affectation partielle $\mathcal{A}$ est définie comme l'ensemble des domaines courants de toutes les variables. Le domaine courant $\mathcal{D}(x)$ d'une variable x est toujours un sous-ensemble de son domaine initial $\mathcal{D}_0(x)$ (inclusion non stricte). On note $var(\mathcal{A})$ l'ensemble des variables instanciées, c'est-à-dire dont le domaine est réduit à un élément. Une affectation est dite partielle si $var(\mathcal{A}) \subsetneq \mathcal{X}$ ou totale si $var(\mathcal{A}) = \mathcal{X}$. Une affectation $\mathcal{A}'$ restreint une affectation partielle $\mathcal{A}$, noté $\mathcal{A}' \subseteq \mathcal{A}$, si le domaine de n'importe quelle variable x dans $\mathcal{A}'$ est un sous-ensemble de son domaine dans $\mathcal{A}$. Les valeurs minimum et maximum du domaine d'une variable entière x sont notées respectivement $\min(x)$ et $\max(x)$. Les notations $\min(x) \leftarrow v$ et $\max(x) \leftarrow v$ représentent respectivement les réductions du domaine à $\mathcal{D}(x) \cap [v, +\infty[$ et $\mathcal{D}(x) \cap]-\infty, v]$. La notation $x \not\leftarrow v$ indique la suppression de la valeur v du domaine $\mathcal{D}(x)$.
Une affectation viole une contrainte si toutes ses variables sont instanciées et que la relation associée à la contrainte n'est pas vérifiée. Une affectation est consistante si elle ne viole aucune contrainte et inconsistante dans le cas contraire. Une solution d'un CSP est donc une affectation totale consistante.
Résoudre un CSP consiste à exhiber une unique solution ou à montrer qu'aucune solution n'existe (le problème est irréalisable). *Prouver la consistance d'un CSP est un problème NP-complet mais exhiber une solution est un problème NP-difficile dans le cas général.* On peut également être intéressé par la détermination de plusieurs ou toutes les solutions d'un problème.

2.1.2 Problème d'optimisation sous contraintes

Dans de nombreux cas, des relations de préférence entre les solutions d'un CSP existent eu égard aux critères fixés par le décideur. L'optimisation consiste alors rechercher des solutions optimales d'un CSP au regard des relations de préférence du décideur. Nous nous intéresserons uniquement à des problèmes d'optimisation combinatoire monocritère, c'est-à-dire lorsque l'ensemble des solutions est discret et qu'il y a un critère d'optimalité unique. Ce critère d'optimalité est généralement associé à la maximisation/-minimisation d'une fonction objectif d'un sous-ensemble de variables vers les entiers.
Un problème d'optimisation sous contraintes (COP) est un CSP augmenté d'une fonction objectif f. Cette fonction est souvent modélisée par une variable dont le domaine est défini par les bornes supérieures et inférieures de f.

2.1.3 Exemple : le problème des n reines

Au cours de ce chapitre, nous illustrerons divers concepts sur le problème des n *reines*. Le but de ce problème, inspiré du jeu d'échec, est de placer n reines sur un échiquier de dimension $n \times n$ de manière à ce qu'aucune ne soit en prise. Deux reines sont en prises si elles sont sur la même ligne, la même colonne ou la même diagonale. Un échiquier classique comporte 8 lignes et 8 colonnes. Ce problème désormais classique est devenu une référence de mesure de performance des systèmes grâce à un énoncé simple masquant sa difficulté. Un modèle classique sans contraintes globales est défini dans la figure 2.1. En observant que deux reines ne peuvent pas être placées sur la même colonne, on peut imposer que la reine i soit sur la colonne i. Ainsi, la variable l_i de domaine $\{1, \ldots, n\}$ représente la ligne où est placée la reine dans la colonne i. Les contraintes (2.1) imposent que les reines soient sur des lignes différentes alors que les contraintes (2.2) et (2.3) imposent que deux reines soient placées sur des diagonales différentes.

$$l_i \neq l_j \qquad 1 \leq i < j \leq n \tag{2.1}$$
$$l_i \neq l_j + (j - i) \qquad 1 \leq i < j \leq n \tag{2.2}$$
$$l_i \neq l_j - (j - i) \qquad 1 \leq i < j \leq n \tag{2.3}$$

FIGURE 2.1 – Exemple de CSP : n reines.

La figure 2.2 montre plusieurs affectations partielles pour le problème des 4 reines dans lesquelles le placement d'une reine sur l'échiquier est représenté par un cercle dans la case correspondante. Les reines sont ordonnées de gauche à droite et les positions de haut en bas. On peut observer en figure 2.2 que l'affectation partielle $\{l_1 \leftarrow 1, l_2 \leftarrow 3, l_3 \leftarrow 2\}$ n'est pas consistante car elle viole une des contraintes (2.2). Au contraire, les affectations totales $\{l_1 \leftarrow 2, l_2 \leftarrow 4, l_3 \leftarrow 1, l_4 \leftarrow 3\}$ et $\{l_1 \leftarrow 3, l_2 \leftarrow 1, l_3 \leftarrow 4, l_4 \leftarrow 2\}$ sont les deux solutions symétriques des 4 reines.

Si l'on définit un COP à partir du problème des n reines en définissant la fonction objectif $min(l_1)$, alors le problème admet une unique solution optimale : $\{l_1 \leftarrow 2, l_2 \leftarrow 4, l_3 \leftarrow 1, l_4 \leftarrow 3\}$.

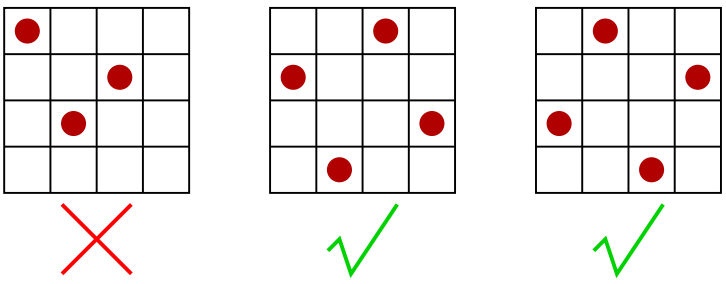


FIGURE 2.2 – Exemples d'affectation : 4 reines.

2.2 Méthodes de résolution

Les méthodes de résolution des CSPs sont génériques, c'est-à-dire qu'elles ne dépendent pas de l'instance à résoudre. Cependant, des techniques dédiées améliorent la résolution de différentes classes de problèmes. Dans le contexte d'un CSP statique et discret, nous discuterons de la réduction de l'espace de recherche par des techniques de consistance couplées si nécessaire à un algorithme de recherche arborescente.
Dans un premier temps, nous présentons deux algorithmes de recherche arborescente dont le comportement peut être amélioré par des techniques de consistance ou des contraintes globales. Ensuite, nous décrivons plusieurs algorithmes avancés, prospectifs ou rétrospectifs, dans le contexte d'une recherche complète par séparation et évaluation.

2.2.1 Algorithmes simples de recherche

L'algorithme *generate-and-test* énumère les affectations totales et vérifie leur consistance, c'est-à-dire qu'aucune contrainte n'est violée. En général, les affectations sont énumérées grâce à une recherche arborescente qui instancie itérativement les variables. Cet algorithme ne réveille les contraintes que lorsqu'une affectation est totale, mais considère par contre un nombre exponentiel d'affectations le rendant impraticable même pour des problèmes de petite taille. Des résultats de complexité ont montré que prouver la satisfiabilité d'un CSP était un problème NP-complet mais qu'exhiber une solution admissible ou optimale était un problème NP-difficile.
L'algorithme simple retour arrière (*backtrack*) [10] étend progressivement l'affectation vide en instanciant une nouvelle variable à chaque étape. L'algorithme vérifie alors que l'affectation partielle étendue est consistante avec les contraintes du problème. Dans le cas contraire, la dernière instanciation faite est remise en cause et l'on effectue une nouvelle instanciation. De cette manière, l'algorithme construit un arbre de recherche dont les nœuds représentent les affectations partielles testées. Cet algorithme teste l'ensemble des affectations possibles de manière implicite, c'est-à-dire sans les générer toutes. Le nombre d'affectations considéré est ainsi considérablement réduit, mais en revanche, la consistance des contraintes est vérifiée à chaque affectation partielle.

La figure 2.3 illustre la résolution du problème des 4 reines par l'algorithme *backtrack* qui place à chaque point de choix la première reine libre en partant de la gauche à la première position disponible en partant du haut. À chaque point de choix, l'affectation partielle courante est étendue en instanciant les variables et les valeurs selon l'ordre lexicographique. Dans ce cas précis, toutes les solutions symétriques sont éliminées en imposant que la première reine (à gauche) soit placée dans la partie haute de l'échiquier. Remarquez que la recherche continue après la découverte de la première solution puisqu'on cherche toutes les solutions à une symétrie près. L'algorithme *generate-and-test* consiste à déployer l'arbre entier.

La découverte redondante d'inconsistance locale due à la perte d'information sur l'inconsistance d'affectation partielle dégrade les performances de l'algorithme *backtrack*. Nous discuterons donc de l'utilisation active des contraintes pour supprimer les inconsistances, des algorithmes prospectifs qui anticipent les prochaines affectations pour réduire les domaines, et rétrospectifs qui utilisent les conflits pour remettre en cause les choix antérieurs.

2.2.2 Filtrage et propagation des contraintes

Le filtrage et la propagation des contraintes [10] permettent d'améliorer les capacités de résolution des CSP lors de la construction d'un arbre de recherche. Une fonction `revise()` associée à chaque contrainte réalise le filtrage des domaines, c'est-à-dire qu'elle supprime les valeurs inconsistantes des domaines de ces variables. Différents niveaux de consistance locale existent pour une même contrainte. La propagation calcule un point fixe global qui est atteint lorsque les techniques de consistance locale ne peuvent plus réaliser aucune inférence. En général, les algorithmes de consistance sont incomplets, c'est-à-dire qu'ils ne retirent pas toutes les valeurs inconsistantes des domaines.

Les opérations (2.4) et (2.5) définissent une fonction `revise()` pour la contrainte arithmétique $x \leq y$. Supposons que les domaines de x et y sont égaux à $\{1, 2, 3\}$, alors l'opération (2.4) réduit le domaine de x à $\{1, 2\}$ tandis que l'opération (2.5) réduit le domaine de y à $\{2, 3\}$. La fonction `revise()` a atteint

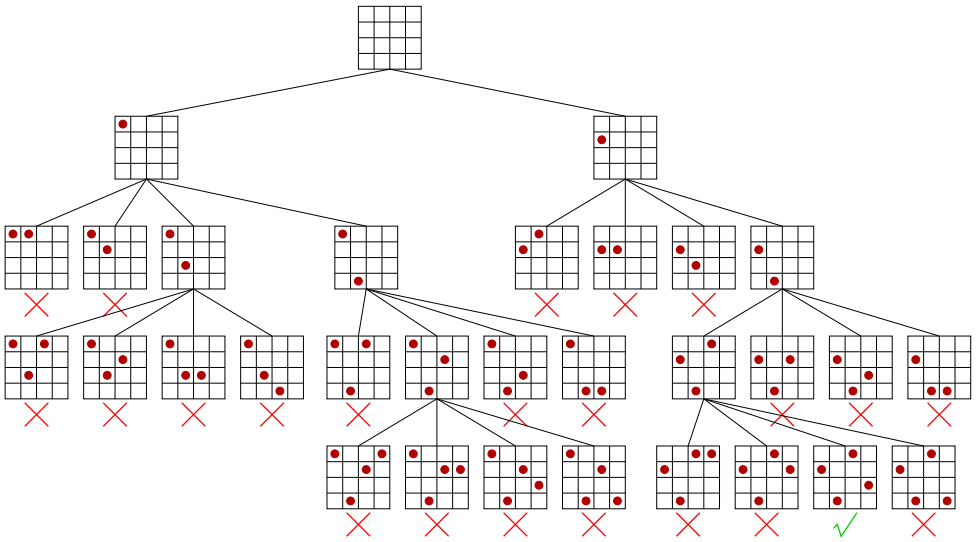


FIGURE 2.3 – Algorithme *backtrack* : 4 reines.

son point fixe puisqu'aucune règle ne peut plus produire d'inférence. Dans de nombreux cas, la fonction `revise()` doit être appelée plusieurs fois avant d'atteindre son point fixe.

$$\max(x) \leftarrow \max(y) - 1 \tag{2.4}$$
$$\min(y) \leftarrow \min(x) + 1 \tag{2.5}$$

Nous rappelons les principales techniques de consistance. Un CSP est dit consistant de nœud si pour toute variable x et pour toute valeur $v \in \mathcal{D}(x)$, l'affectation partielle $x \leftarrow v$ satisfait toutes les contraintes unaires, c'est-à-dire dans lesquelles x est l'unique variable non instanciée. Une contrainte est dite arc-consistante si pour chaque valeur de chaque variable x, il existe une affectation des autres variables telle que la contrainte soit satisfaite. Un CSP est dit arc-consistant lorsque toutes ses contraintes sont arc-consistantes. Néanmoins, il est insuffisant d'effectuer une révision unique par contrainte pour atteindre le point fixe global. Par conséquent, l'algorithme AC-1 révise toutes les contraintes jusqu'à ce qu'aucun domaine n'ait changé. L'algorithme AC-3 utilise une file de contraintes à réviser dans laquelle il ajoute les contraintes portant sur une variable dont le domaine a changé. En pratique, les algorithmes à partir d'AC-3 affinent le réveil des contraintes en fonction d'évènements sur les domaines définis par le solveur (réduction, suppression, instanciation ...). Toutefois, l'arc-consistance ne détecte pas les inconsistances dues à un ensemble de contraintes comme nous le verrons dans la section suivante.
Lorsque les domaines des variables sont trop grands, la mémoire requise pour stocker l'appartenance ou non de chaque valeur devient problématique. De la même manière, l'application des techniques de consistance pour chaque couple de variable et de valeur peut dégrader considérablement la vitesse de propagation par rapport à la réduction effective des domaines. On utilise alors la consistance de bornes qui consiste à raisonner sur la valeur minimale et maximale que les variables peuvent prendre. Pour certaines contraintes, la consistance de borne est très proche, voire égale à la consistance d'arc, notamment pour la contrainte $x \leq y$ discutée ci-dessus.

Ainsi, à chaque nœud de l'arbre de recherche, c'est-à-dire à chaque instanciation d'une variable, l'algorithme de recherche applique des techniques de consistance locale, propre à chaque contrainte, qui retirent des valeurs inconsistantes des domaines. À leur tour, les modifications des domaines peuvent inférer de nouvelles inconsistances. Ce mécanisme, appelé propagation, continue jusqu'à ce qu'un point fixe global soit atteint. L'algorithme fait alors un nouveau choix de variable et de valeur et teste la nouvelle instanciation. Si durant le filtrage d'une contrainte, le domaine d'une variable devient vide, alors l'instanciation courante est inconsistante. Le nœud est fermé et l'algorithme passe au nœud suivant s'il existe. Cette opération permet d'agir sur les domaines de toutes les variables d'un CSP durant la construction des affectations partielles réduisant ainsi le nombre de nœuds composant l'arbre.

2.2.3 Contraintes globales

La modélisation des problèmes complexes est facilitée par l'utilisation de contraintes hétérogènes agissant indépendamment sur de petits ensembles de variables. Toutefois, la détection locale des inconsistances affaiblit la réduction des domaines. Les contraintes globales corrigent partiellement ce comportement en utilisant l'information sémantique issue de raisonnements sur des sous-problèmes. Elles permettent généralement d'augmenter l'efficacité du filtrage ou de réduire les temps de calcul.

Par exemple, l'arc-consistance ne détecte pas l'inconsistance globale du CSP présenté dans la figure 2.4. En effet, l'inconsistance est issue des trois contraintes qui imposent que les trois variables prennent des

$$x \neq y \qquad x \neq z \qquad z \neq y$$
$$\mathcal{D}_0(x) = \mathcal{D}_0(y) = \mathcal{D}_0(z) = \{0, 1\}$$

FIGURE 2.4 – Exemple de CSP où l'arc-consistance est incomplète.

valeurs distinctes alors que l'union de leurs domaines ne contient que deux valeurs. La contrainte globale `allDifferent` $(x_1, \ldots, x_n)$ qui impose que ses variables prennent des valeurs distinctes détecte cette inconsistance triviale. Ses algorithmes de filtrage reposent sur le calcul de couplages maximaux dans un graphe biparti dont les deux ensembles de sommets représentent respectivement les variables et les valeurs alors que les arêtes représentent les instanciations possibles [11].

Un second modèle classique pour le problème des reines défini dans la figure 2.5 utilise des contraintes globales `allDifferent` pour représenter les cliques d'inégalités binaires. On introduit généralement les variables auxiliaires x_i et y_i par le biais des contraintes de liaison (2.6). Les variables auxiliaires correspondent aux projections sur la première colonne de la reine i en suivant les diagonales. La contrainte (2.7) impose que les reines soient sur des colonnes différentes alors que les contraintes (2.8) et (2.9) imposent que deux reines soient placées sur des diagonales différentes.

$$x_i = l_i - i \qquad y_i = l_i + i \qquad 1 \leq i \leq n \tag{2.6}$$
$$\texttt{allDifferent}\,(l_1, \ldots, l_n) \tag{2.7}$$
$$\texttt{allDifferent}\,(x_1, \ldots, x_n) \tag{2.8}$$
$$\texttt{allDifferent}\,(y_1, \ldots, y_n) \tag{2.9}$$

FIGURE 2.5 – Exemple de CSP utilisant des contraintes globales : n reines.

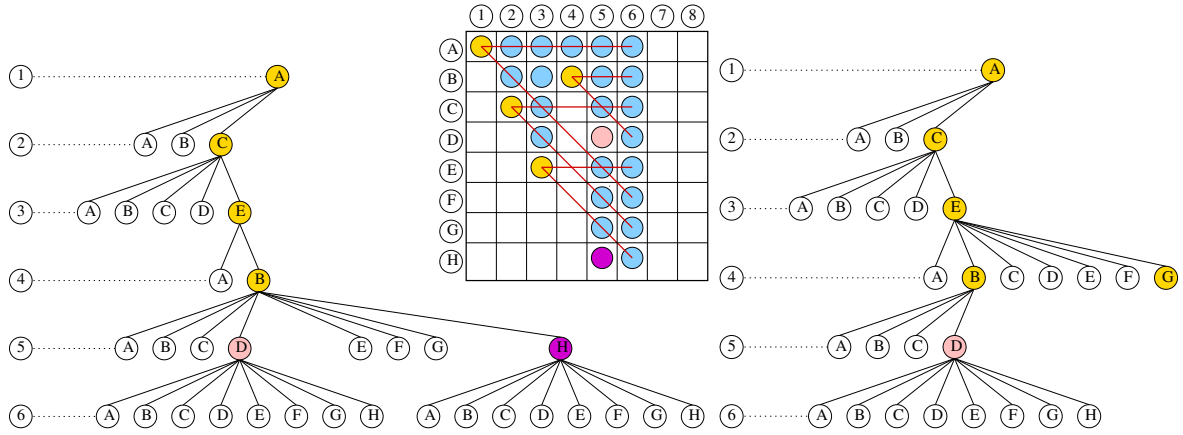
Beldiceanu et Demassey [12] proposent une classification complète des contraintes globales accompagnée d'une description de leurs algorithmes de filtrage dans laquelle figurent la plupart des contraintes discutées dans cette thèse.

2.2.4 Algorithmes avancés de recherche

Une technique prospective simple pour anticiper les effets d'une instanciation est nommée *forward checking*. On vérifie que les variables non instanciées peuvent chacune prendre une valeur consistante lorsque l'affectation partielle courante est étendue par l'instanciation d'une nouvelle variable. Les techniques de *look-ahead* sont plus lentes, mais procurent un meilleur filtrage basé sur l'arc-consistance. L'arc-consistance est appliquée sur toutes les variables non instanciées pour chaque affectation étendue. Ainsi, les techniques de *forward checking* considèrent une seule variable non instanciée à la fois pour la consistance alors que celles de *look-ahead* considèrent une paire de variables non instanciées.

D'un autre côté, les techniques rétrospectives remettent en cause l'affectation partielle lorsqu'une inconsistance est détectée lors de la propagation. Le *backtrack chronologique* consiste simplement à remettre en cause le dernier choix. Cet algorithme est sensible au phénomène de *thrashing* où des inconsistances

dues à un nombre restreint d'instanciations sont redécouvertes de manière redondante lors de l'exploration de l'arbre de recherche. La figure 2.6(a) illustre ce phénomène sur le problème des 8 reines lors de la remise en cause du placement de la reine 5 (colonnes D et H) car les conflits sur le placement de la reine 6 sont indépendants de cette décision.



(a) Thrashing : backtrack chronologique. (b) Conflict directed backjumping.

FIGURE 2.6 – Algorithmes de recherche rétrospectifs : 8 reines.

Le *conflit directed backjumping* ou *intelligent backtracking* [13] consiste à calculer une explication à un backtrack puis à remettre en cause le choix le plus récent de cette explication. Chaque échec sur le choix d'une valeur est expliqué par les précédents choix qui entrent en conflit. Si toutes les valeurs d'un domaine ont été testées sans succès, l'explication de cet échec est l'union des explications des valeurs du domaine. Par conséquent, le placement de la reine 4 est remis en cause dans la figure 2.6(b) lors du backtrack après la tentative de placement de la reine 6 car le conflit est expliqué par les choix portant sur les reines 1, 2, 3 et 4.
Le *dynamic backtracking* [14] utilise un mécanisme similaire mais sans remettre en cause les choix indépendants de l'explication. Par exemple, la prochaine décision du *dynamic backtracking* au nœud G de la reine 4 dans la figure 2.6(b) est le placement de la reine 5 dans la colonne D.
Le *nogood recording* [15] consiste à mémoriser les causes des conflits puis à utiliser des techniques de propagation inspirées des problèmes de satisfiabilité booléenne (SAT).

2.3 Stratégies de recherche

Les techniques de consistance étant incomplètes, les algorithmes de recherche résolvent les disjonctions restantes par des méthodes de séparation. Le nombre de nœuds et le temps de résolution d'un modèle peuvent varier considérablement en fonction de la méthode de séparation et des heuristiques de sélection employées.

2.3.1 Méthodes de séparation

À chaque création d'un nœud de l'arbre de recherche, l'algorithme crée un point de choix, c'est-à-dire qu'il considère successivement la résolution de sous-problèmes disjoints dans chaque branche. Dans un point de choix binaire, on peut considérer l'ajout d'une contrainte C dans la première branche et $\neg C$ dans la seconde grâce à la réversibilité des contraintes. Par exemple, la technique de *variable ordering* impose une relation d'ordre sur une paire de variables x et y : $x < y \vee x \geq y$. Cependant, on peut se limiter à considérer des points de choix opérant des restrictions sur le domaine d'une unique variable non instanciée. En effet, le mécanisme de réification d'une contrainte, c'est-à-dire l'association d'une variable booléenne indiquant la satisfaction de la contrainte, permet la transformation d'un point de choix sur des contraintes en un ou plusieurs autres portant sur des variables. De plus, la plupart des algorithmes

rétrospectifs ou techniques d'apprentissage utilisent des variables car leur domaine est restauré lors d'un retour arrière alors que les contraintes sont généralement supprimées. Les techniques standards opèrent généralement une restriction sur le domaine d'une seule variable $x \in \mathcal{X}$ choisie par une heuristique de sélection de variable.
Par exemple, un point de choix appliquant le *standard labeling* considère l'affectation d'une valeur $v \in \mathcal{D}(x)$ à la variable ou bien sa suppression : $x = v \vee x \neq v$. De la même manière, un point de choix appliquant le *domain splitting* restreint la variable x à des valeurs inférieures ou supérieures à une valeur seuil $v \in \mathcal{D}(x) : x < v \vee x \geq v$. La valeur v est choisie par une heuristique de sélection de valeur. On peut définir des branchements n-aires à partir des précédents en testant une valeur différente dans chaque branche (*standard labeling*) et en restreignant les valeurs de x à des intervalles disjoints dans chaque branche (*domain splitting*).
On appelle variables de décision, un sous-ensemble de variables dont l'instanciation provoque une affectation totale par propagation. Les méthodes de séparation peuvent limiter la profondeur de l'arbre en se restreignant à un sous-ensemble de variables de décision. Par exemple, les variables l_i du modèle des n reines en figure 2.5 sont des variables de décision. Cependant, on peut leur substituer les variables auxiliaires x_i et y_i qui sont aussi des variables de décision. Par défaut, toutes les variables du problème sont des variables de décision.
La programmation par contraintes offre donc un cadre flexible pour la définition de méthodes de séparation de par la structure des points de choix et l'utilisation d'heuristiques de sélection évoquée ci-dessous.

2.3.2 Heuristiques de sélection

Les heuristiques de sélection sont des règles qui déterminent (a) l'ordre suivant lequel on va choisir les variables pour les instancier, ainsi que (b) l'ordre suivant lequel on va choisir les valeurs pour les variables. Une bonne heuristique de sélection peut avoir un impact important sur la résolution d'un problème de décision et accélérer l'obtention de solutions (la détection d'échecs en cas de problèmes insolubles) pendant l'exploration de l'espace de recherche. Les algorithmes de résolution qui utilisent des heuristiques de sélection assurent, dans la plupart des cas, une résolution plus rapide que celle obtenue par des algorithmes sans heuristique de sélection. Une heuristique est statique si l'ordre est préalablement fixé ou dynamique s'il évolue au cours de la résolution. Les heuristiques de sélection sur des variables ensemblistes ne seront pas abordées dans le cadre de cette thèse.

2.3.2.1 Sélection de variable

Les heuristiques de sélection de variable ont souvent une influence critique sur la forme et la taille de l'arbre de recherche. Cette influence est toutefois plus restreinte lorsqu'on cherche toutes les solutions d'un CSP. De nombreuses heuristiques observent le principe *first-fail* énoncé par Haralick et Elliott [16] : « To succeed, try first where you are most likely to fail » (Pour réussir, essaie d'abord là où il est le plus probable d'échouer). Les heuristiques dynamiques exploitent souvent les informations sur les domaines et degrés des variables. On retrouve entre autres les heuristiques suivantes.

lex ordonne les variables selon l'ordre lexicographique (statique).

random sélectionne aléatoirement une variable non instanciée.

min-domain ou dom sélectionne la variable non instanciée qui a le plus petit domaine de valeurs.

degree ou deg l'heuristique du degré [17] ordonne les variables selon leurs degrés décroissants.

dynamic-degree ou ddeg l'heuristique du degré dynamique sélectionne la variable qui a le degré dynamique le plus élevé. Le degré dynamique est calculé pour l'affectation partielle courante.

weighted-degree ou wdeg l'heuristique du degré pondéré [18] se base sur l'apprentissage du poids de chaque contrainte en exploitant les étapes précédentes de la résolution. Le poids d'une contrainte est égal au nombre d'échecs provoqués par celle-ci. Elle sélectionne la variable dont la somme des degrés dynamiques pondérés est maximale.

dom/deg, dom/ddeg et dom/wdeg il s'agit de combinaisons des heuristiques précédentes dans laquelle on sélectionne la variable dont le quotient de la taille du domaine sur le degré est minimal.

impact l'impact d'une variable ou d'une affectation mesure son influence sur la réduction de l'espace de recherche. Leur apprentissage se base sur la surveillance des réductions de domaine pendant

la recherche. La recherche basée sur les impacts [19] sélectionne la variable d'impact maximal et l'instancie à sa valeur d'impact minimal.

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.

Des techniques de redémarrage peuvent souvent être avantageusement combinées aux heuristiques par apprentissage (*wdeg*, *dom/wdeg*, *impact*) ou intégrant un élément de randomisation. Même si les différentes heuristiques de sélection sont en général incomparables entre elles, dans de nombreux cas, notamment sur des problèmes structurés, *dom/wdeg* fournit de très bons résultats.

2.3.2.2 Sélection de valeur

Une heuristique de sélection de valeur détermine la prochaine valeur du domaine à tester. Elle définit donc l'ordre d'exploration des branches d'un point de choix et dépend généralement du problème. Imaginons l'existence d'un oracle fournissant une valeur consistante pour chaque point de choix sur une variable, alors l'algorithme de recherche trouvera la première solution sans retour arrière quel que soit l'ordre d'énumération des variables. Ces heuristiques observent souvent le principe *succeed-first* qui consiste à choisir la valeur ayant la plus grande probabilité d'appartenir à une solution. L'heuristique **minVal** sélectionne la plus petite valeur du domaine, **maxVal** sélectionne la plus grande, alors que **randVal** sélectionne aléatoirement une valeur. Lorsque les domaines des variables sont énumérés, on peut appliquer l'heuristique **midVal** qui sélectionne la valeur médiane du domaine.

2.3.2.3 Exemple et discussion

Nous montrons sur un exemple l'influence des heuristiques de sélection sur la résolution du problème des 4 reines par l'algorithme *backtrack* avec un branchement n-aire par *standard labeling*. La figure 2.3 en page 14 représente l'arbre de recherche obtenu avec les heuristiques de sélection *lex* et *minVal*. La figure 2.7 représente l'arbre obtenu avec une heuristique de sélection de variable imposant un ordre statique $\{l_1, l_2, l_4, l_3\}$ combinée à une heuristique de sélection de valeur *maxVal*. L'avantage relatif à la découverte plus précoce de la solution est contrebalancé par l'augmentation de la taille de l'arbre établissant son unicité.

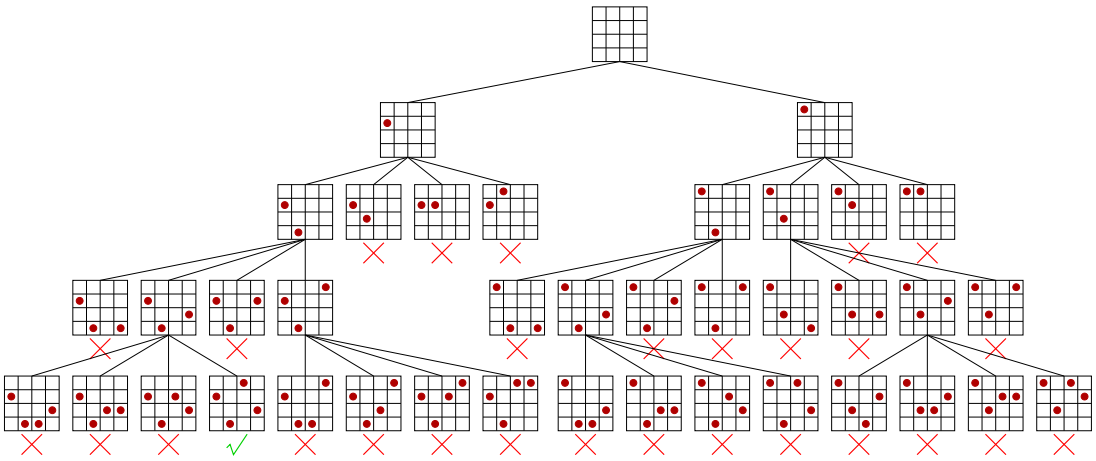


FIGURE 2.7 – Influence des heuristiques de sélection sur l'algorithme *backtrack* : 4 reines.

Il est important de remarquer que les heuristiques disposent de moins d'information au début de la recherche alors que les premiers points de choix ont une importance cruciale. Cette situation peut mener aux situations de *thrashing* décrites précédemment puisque les heuristiques ne sont pas infaillibles.

Cependant, l'exploration ne devrait pas trop s'éloigner des choix de l'heuristique lorsque celle-ci est très bien adaptée. La recherche en profondeur, utilisée par exemple dans l'algorithme de simple retour arrière, s'éloigne rapidement des choix initiaux de l'heuristique. Pour pallier ce problème, la *limited discrepancies search* (LDS) explore l'arbre de manière à remettre en cause un nombre minimum des choix initiaux de l'heuristique [20]. La figure 2.8 compare l'ordre d'exploration des branches d'une recherche en profondeur (à gauche) à LDS (au milieu) et une de ses variantes *depth-bounded discrepancies search* (DBDS). Les

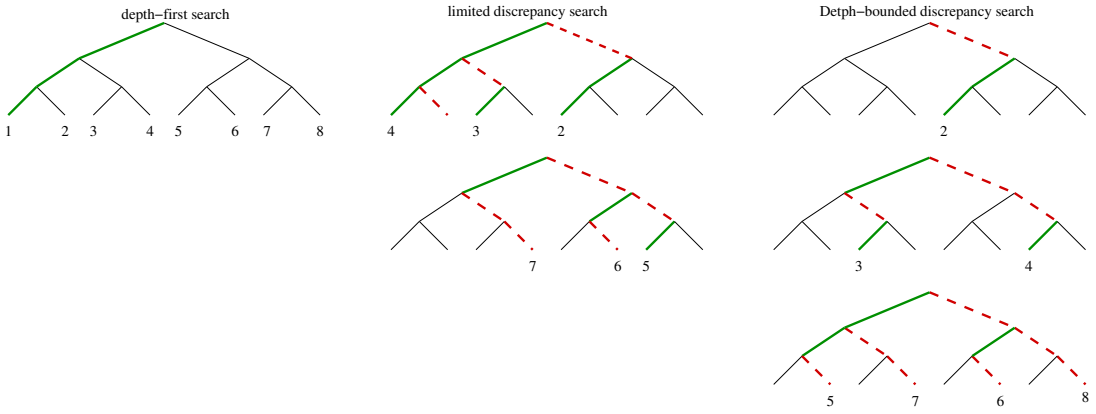


FIGURE 2.8 – Ordre d'exploration des branches par différentes recherches arborescentes.

branches en gras indiquent le respect des choix initiaux et celles en pointillés indiquent leur remise en cause. LDS et DBDS remettent en cause prioritairement les choix les plus précoces considérés moins fiables. Au contraire de LDS, DBDS cherche d'abord à minimiser la profondeur du dernier choix remis en cause plutôt que leur nombre.

2.4 Procédures d'optimisation

En pratique, résoudre un problème d'optimisation sous contraintes (COP) consiste à trouver une ou les solutions du CSP dont la valeur de la variable objectif est minimale ou maximale. Les extensions dédiées à l'optimisation reposent sur la résolution d'une série de problèmes de satisfaction de contraintes. La répartition et le nombre (au pire cas) de sous-problèmes satisfiables et insatisfiables caractérisent ces procédures. L'évaluation initiale de l'objectif a une influence majeure sur la construction de la série qui dépend également de l'algorithme de résolution des sous-problèmes. Le choix d'une procédure d'optimisation repose souvent sur l'analyse du modèle et des fonctionnalités du solveur de contraintes. La combinaison judicieuse d'une procédure et d'un algorithme de résolution des sous-problèmes est un élément clé d'une approche en optimisation sous contraintes, notamment pour les problèmes traités dans cette thèse.
Dans cette section, nous présentons d'abord deux procédures classiques bottom-up et top-down respectivement orientées vers l'insatisfiabilité et la satisfiabilité des sous-problèmes. Ensuite, nous décrivons plusieurs variantes complètes et incomplètes qui seront discutées ou utilisées ultérieurement dans ce manuscrit. Nous avons jugé intéressant de regrouper les descriptions de ces procédures, mais il est possible de les lire au cas par cas. Pour chaque procédure, un tableau récapitulera le nombre de sous-problèmes satisfiables et insatisfiables en fonction de la satisfiabilité du COP.

Sans perte de généralité, nous supposons que nous minimisons une fonction objectif représentée par une variable entière obj dont le domaine initial $\mathcal{D}_0(obj) = [lb, ub[$ est un intervalle fini non vide. Si problème est satisfiable, la valeur de l'optimum est notée opt. Une procédure d'optimisation réduit itérativement le domaine de l'objectif jusqu'à atteindre une condition d'arrêt : $lb = ub$. Lorsque le problème est satisfiable, la nouvelle valeur ub appartient à $\mathcal{D}_0$. La fonction `solve`(x, y) applique un algorithme de recherche complet pour résoudre le CSP où le domaine de l'objectif est réduit à l'intervalle $[x, y[$. La résolution s'achève à la découverte d'une solution et la fonction `solve`(x, y) renvoie la valeur de la fonction objectif. Lorsque le sous-problème est insatisfiable ou que la résolution est interrompue, la fonction renvoie la

valeur null.

À titre d'exemple, nous considérerons un COP où $\mathcal{D}_0(obj) = [0, 16[$ et dont les coûts des solutions appartiennent à l'ensemble $\{9, 11, 13, 15\}$. Dans les figures 2.9, 2.10, 2.11 et 2.12, le domaine initial est représenté par un tableau où les cellules contenant des valeurs correspondant à une solution sont indiquées par une étoile. La résolution d'un sous-problème par la fonction `solve(`x, y`)` est représentée par une flèche dont le numéro correspond à la position du sous-problème dans la série. Lorsque le sous-problème est insatisfiable, la flèche part du coin supérieur gauche de la cellule contenant l'ancienne valeur lb et pointe vers le coin supérieur gauche de la cellule contenant la nouvelle valeur lb (orientation gauche – droite). Lorsque le sous-problème est satisfiable, la flèche part du milieu de la cellule contenant l'ancienne valeur ub et pointe vers le milieu de la cellule contenant la nouvelle valeur ub (orientation droite – gauche).

2.4.1 Procédure bottom-up

La procédure bottom-up incrémente la borne inférieure lb de la fonction objectif lorsque le problème de satisfaction de contraintes `solve(`$lb, lb + 1$`)` est insatisfiable jusqu'à ce qu'une solution optimale soit trouvée. Lorsque le COP est satisfiable, la procédure résout $opt - lb$ problèmes insatisfiables avant de résoudre un unique problème satisfiable fournissant une solution optimale. Le COP est implicitement considéré comme étant satisfiable en regard de l'efficacité amoindrie de la procédure dans le cas contraire. En effet, lorsque le COP n'est pas satisfiable, il est plus efficace de résoudre directement le problème `solve(`lb, ub`)` plutôt que $ub - lb$ sous-problèmes disjoints. La construction des arbres de recherche pour des problèmes voisins peut dégrader les performances à cause de la propagation redondante et de la nécessité de recréer les points de choix. En fait, L'efficacité de bottom-up dépend de la qualité de la borne inférieure initiale et de la capacité de l'algorithme de résolution à prouver l'insatisfiabilité des sous-problèmes, généralement grâce à des techniques avancées de propagation et de filtrage. Il peut être intéressant d'utiliser une borne inférieure dédiée fournissant une garantie, de relever le niveau de consistance ou de renforcer la propagation par des techniques prospectives pour réduire la taille des arbres de recherche et par conséquent les redondances. Par contre, l'algorithme dispose d'informations plus précises car la valeur de l'objectif est fortement contrainte.
Nous verrons que le nombre de sous-problèmes traités par bottom-up sur l'exemple de la figure 2.9 est plus élevé que celui des autres procédures. En pratique, la difficulté moindre des premières étapes peut compenser le nombre plus élevé de sous-problèmes.

Procédure bottom-up(lb,ub)

tant que $lb < ub$ **faire**
  si `solve(`lb,$lb+1$`)` = null **alors** $lb \leftarrow lb + 1$;
  sinon $ub \leftarrow lb$;

	#sous-problèmes	
COP	$\#SAT$	$\#UNSAT$
SAT	1	$opt - lb$
UNSAT	0	$ub - lb$

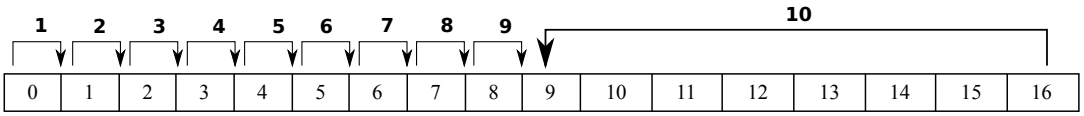


FIGURE 2.9 – Description de la procédure d'optimisation bottom-up.

2.4.2 Procédure top-down

Le succès des métaheuristiques s'explique entre autres par la découverte rapide de très bonnes solutions sans se préoccuper d'atteindre la preuve d'optimalité. Un objectif monocritère représente rarement la complexité d'une situation réelle et le bon sens recommande d'examiner des solutions diverses dont la qualité est jugée satisfaisante. Pour satisfaire cette demande, la stratégie top-down résout une série de problèmes de satisfaction dont les solutions améliorent la borne supérieure courante jusqu'à ce que le sous-problème devienne insatisfiable. Elle permet ainsi la découverte rapide de « bonnes » solutions alors que bottom-up fournit une unique solution optimale à la fin du calcul. La procédure débute par la recherche d'une première solution réalisable du COP par l'application de `solve(`lb,ub`)`. Lorsqu'une solution existe,

une boucle construit et résout des sous-problèmes de satisfaction dont les solutions améliorent la borne supérieure courante. Pour ce faire, le solveur doit proposer un service de coupe par le biais de la fonction `postDynamicCut`. Une coupe est une contrainte dont l'ajout est permanent contrairement à une contrainte qui est supprimée lors d'un retour arrière. Après la découverte d'une solution, une coupe restreint le domaine de l'objectif à des valeurs inférieures à la nouvelle borne supérieure. La recherche arborescente est ensuite reprise, grâce à la méthode `resumeSolve()`, à partir du dernier point de choix réalisable après l'ajout de la coupe. Contrairement à bottom-up, la structure des sous-problèmes est exploitée pour construire un unique arbre de recherche, au prix de services supplémentaires (interruption – reprise de la recherche, ajout de coupes). La procédure top-down résout un unique sous-problème lorsque le COP est insatisfiable. Dans le cas contraire, le nombre de sous-problèmes dépend des solutions améliorantes découvertes. La borne supérieure influence la réduction des domaines et par conséquent les informations utiles à l'algorithme de recherche. Ainsi, le nombre maximal de sous-problèmes $ub - opt$. Au contraire, la borne inférieure de l'objectif a une influence marginale sur la procédure mais peut éventuellement faciliter la preuve d'optimalité, c'est-à-dire la résolution du dernier sous-problème insatisfiable.
On peut constater sur l'exemple de la figure 2.10 que top-down ne visite pas toutes les solutions avant de découvrir l'optimum. En effet, la résolution du troisième sous-problème fournit la solution optimale avant la solution améliorante. Ce comportement illustre l'influence des heuristiques de sélection qui guident quelquefois la recherche vers de bonnes solutions même lorsque l'objectif est faiblement contraint.

Procédure top-down(lb,ub)

```
sol ← solve(lb,ub) ;
tant que sol ≠ null faire
    postDynamicCut(ub ← sol);
    sol ← resumeSolve();
lb ← ub;
```

	#sous-problèmes	
COP	$\#SAT$	$\#UNSAT$
SAT	$\leq ub - opt$	$\min(1, opt - lb)$
UNSAT	0	1

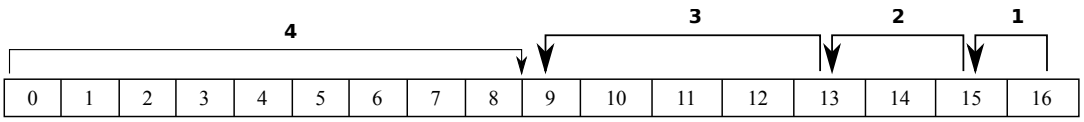


FIGURE 2.10 – Description de la procédure d'optimisation top-down.

2.4.3 Procédure dichotomic-bottom-up

La procédure dichotomic-bottom-up améliore bottom-up grâce à un partitionnement dichotomique du domaine de l'objectif. Elle considère ainsi un nombre restreint de sous-problèmes dont certains fournissent des solutions intermédiaires. À chaque itération, on résout un sous-problème dans lequel la valeur de l'objectif doit appartenir à la première moitié de son domaine courant. On actualise la borne supérieure de l'objectif lorsque le sous-problème fournit une solution et sa borne inférieure dans le cas contraire. Lorsque le COP est insatisfiable, le nombre de sous-problèmes $O(\log_2(ub - lb + 1))$ est considérablement réduit par rapport à bottom-up. Lorsque le COP est satisfiable, la procédure traite au plus $\log_2(ub - opt)$ sous-problèmes insatisfiables et $\log_2(opt - lb) + 1$ sous-problèmes satisfiables. Le nombre de sous-problèmes peut alors dépasser celui de bottom-up. Par exemple, la découverte des solutions intermédiaires freine la résolution si la borne inférieure initiale est égale à l'optimum.
L'exemple de la figure 2.11 illustre le mouvement de va-et-vient lors de la réduction du domaine de l'objectif par dichotomic-bottom-up qui fournit une solution intermédiaire dès le deuxième sous-problème. La découverte de la solution intermédiaire illustre la sensibilité des heuristiques de sélection au domaine de l'objectif, car le deuxième sous-problème de top-down fournit directement la solution optimale.

2.4.4 Procédures incomplètes

L'évaluation initiale de l'objectif joue donc un rôle crucial concernant le nombre de sous-problèmes mais aussi leur résolution. Des méthodes d'approximation peuvent éventuellement améliorer ou garan-

Procédure dichotomic-bottom-up(lb,ub)

tant que $lb < ub$ **faire**
  $sol \leftarrow$ `solve` $(lb, \lceil \frac{lb+ub}{2} \rceil)$;
  si $sol =$ `null` **alors** $lb \leftarrow \lceil \frac{lb+ub}{2} \rceil$;
  sinon $ub \leftarrow sol$;

	#sous-problèmes	
COP	$\#SAT$	$\#UNSAT$
SAT	$\leq \log_2(ub - opt)$	$\leq \log_2(opt - lb + 1)$
UNSAT	0	$\log_2(ub - lb) + 1$

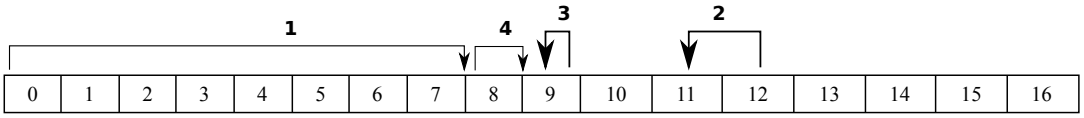


FIGURE 2.11 – Description de la procédure d'optimisation dichotomic-bottom-up.

tir la qualité de cette évaluation initiale. Cependant, elles demandent un effort de compréhension et d'implémentation voire d'adaptation en présence de contraintes additionnelles. Nous présentons plusieurs procédures d'optimisation incomplètes qui affinent l'évaluation initiale de l'objectif.

2.4.4.1 Procédure destructive-lower-bound

La procédure destructive-lower-bound inspirée de bottom-up affine l'évaluation de la borne inférieure sans appliquer de recherche arborescente. Leur principe consiste à fixer une valeur maximale de l'objectif,

Procédure destructive-lower-bound(lb,ub)

tant que $(lb < ub) \wedge$ `isNotFeasible(`lb,$lb+1$`)` **faire**
  $lb \leftarrow lb + 1$;

puis essayer de contredire (détruire) la satisfiabilité du problème ainsi reduit. La fonction `isNotFeasible` applique un test de consistance basé sur la propagation initiale éventuellement complété par différentes techniques prospectives comme du *shaving*. La borne inférieure lb est incrémentée tant que le sous-problème `isNotFeasible` $(lb, lb + 1)$ est prouvé insatisfiable sans retour arrière. Dans le cas contraire, lb est une borne inférieure valide pour le problème. Elle est souvent utilisée pour comparer l'efficacité de différents filtrages car son résultat ne dépend ni de l'algorithme de recherche ni des heuristiques de sélection. Elle permet aussi d'affiner rapidement l'évaluation de la qualité des solutions fournies par la procédure top-down lorsque la preuve d'optimalité n'est pas atteinte, par exemple lors d'une interruption de la recherche (limites de temps, nœuds ...). Par contre, elle est inutile dans le contexte de procédures inspirées de bottom-up qui fournissent une meilleure évaluation de la borne inférieure puisque les preuves sont obtenues par des recherches arborescentes complètes.

2.4.4.2 Procédure destructive-upper-bound

La procédure destructive-upper-bound offre une garantie sur la nouvelle évaluation des bornes inférieure et supérieure de l'objectif. Elle consiste à partitionner le domaine de l'objectif en intervalles $I_k = \left[lb + (2^k - 1), lb + (2^{k+1} - 1)\right[$. Lorsque le problème d'optimisation est insatisfiable, la procédure résout un nombre de sous-problèmes identique à celui de dichotomic-bottom-up mais le partitionnement du domaine de l'objectif est différent. Dans le cas contraire, elle résout $\log_2(opt - lb + 1)$ sous-problèmes insatisfiables améliorant l'évaluation de la borne inférieure et un unique sous-problème satisfiable garantissant que la solution appartient au plus petit intervalle I_k contenant une solution. Les conditions d'arrêt sont différentes puisque la procédure se contente d'affiner l'évaluation de l'objectif et ne garantit pas l'optimalité de la solution. Lorsque le problème est satisfiable, la nouvelle évaluation ub respecte la relation $ub \geq lb$ et, dans le cas contraire, la valeur ub n'est pas modifiée.
La figure 2.12 illustre la résolution du problème en appliquant successivement destructive-upper-bound puis dichotomic-bottom-up (flèches en pointillés). Dans notre exemple, il est préférable de ne pas appliquer la procédure destructive-upper-bound avant la procédure dichotomic-bottom-up. En effet, les

procédures dichotomic-bottom-up (figure 2.11) et destructive-upper-bound (figure 2.11) accomplissent le même nombre d'étapes pour (a) résoudre le problème d'optimisation et (b) améliorer l'évaluation de l'objectif. Par contre, cette combinaison est efficace lorsque la borne supérieure initiale est mauvaise. Par exemple, supposons que l'optimum est égal à 1, la combinaison traiterait deux sous-problèmes alors que dichotomic-bottom-up traiterait au pire cinq sous-problèmes. En effet, la combinaison supprime l'influence de la borne supérieure initiale sur le nombre de sous-problèmes, car la construction des intervalles I_k fournit une garantie, certes faible, sur la nouvelle borne supérieure ($log_2(ub-lb) \leq \log_2(opt-lb+1)+1$). Ainsi, le nombre de sous-problèmes dépend seulement de la différence entre l'optimum et la borne inférieure et non de la différence entre la borne supérieure et inférieure.

Procédure destructive-upper-bound(lb,ub)

$k \leftarrow 0$;
tant que $lb + (2^k - 1) < ub$ **faire**
  $sol \leftarrow$ `solve`$(lb + (2^k - 1), lb + (2^{k+1} - 1))$;
  si $sol =$ `null` **alors** $k \leftarrow k + 1$;
  sinon $ub \leftarrow sol$;**break** ;
$lb \leftarrow lb + (2^k - 1)$;

	#sous-problèmes	
COP	$\#SAT$	$\#UNSAT$
SAT	1	$\log_2(opt - lb + 1)$
UNSAT	0	$\log_2(ub - lb) + 1$

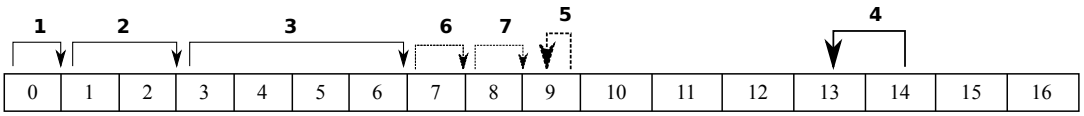


FIGURE 2.12 – Description de la procédure d'optimisation destructive-upper-bound.

2.4.4.3 Autres procédures

Les procédures bottom-up et dichotomic-bottom-up sont incomplètes si l'on impose une limite (temps, nœuds ...) sur la résolution de chaque sous-problème. On peut également limiter la « profondeur dichotomique » de dichotomic-bottom-up, c'est-à-dire le nombre maximal de divisions du domaine de la variable objectif.

2.5 Solveurs de contraintes

Nous citons quelques outils de programmation par contraintes libres ou commerciaux. Cette revue ne se veut pas exhaustive tant les outils disponibles évoluent continuellement. Dans un souci de concision, nous ne citerons que les solveurs dédiés aux problèmes booléens (SAT) et à la programmation mathématique mentionnés dans les chapitres suivants. Les outils présentés diffèrent en plusieurs points : les entités qu'ils sont capables de traiter (entiers, ensembles, objets ...) ; les contraintes et algorithmes de recherche implémentés ; le langage hôte sur lequel il s'appuie. Historiquement, les premiers solveurs étaient des extensions du langage déclaratif de programmation logique `Prolog` [21] dont l'algorithme d'unification est une clé. De nos jours, les extensions de Prolog les plus populaires sont `ECLiPSe` [22] et `SICStus Prolog` [23] qui proposent toutefois des interfaces vers des langages objet tels que Java, .NET, C et C++ ainsi qu'une compatibilité avec les langages d'autres solveurs. Les langages C et C++ ont connu beaucoup de succès grâce à leur rapidité et la gestion optimisée de la mémoire. Les noyaux des solveurs commerciaux `CHIP` [24] et `Ilog CP Optimizer` [25] sont principalement écrits en C/C++ mais leur architecture complexe comporte de nombreuses couches logicielles et interfaces dans d'autres langages. Les solveurs libres `gecode` [26] et `mistral` [27] entièrement écrits en C/C++ proposent des services classiques tout en restant modulaire et rapide. Il existe quelques solveurs libres implémentés en Java comme `choco` [28] et `koalog` [29] malgré des critiques concernant la gestion de la mémoire, notamment le ramasse-miettes (*garbage collector*), et les performances des programmes. Pour répondre à ces critiques, on peut d'abord remarquer que les performances des solveurs dépendent bien plus des technologies embarquées que du langage hôte. Ensuite, leur code source ouvert ainsi que la popularité et la portabilité du langage Java

en font des outils de choix pour la recherche et l'enseignement. Plus récemment, `comet` [30] a proposé un langage de modélisation orienté objet compréhensible par un solveur de contraintes proposant aussi des services de recherche locale et de programmation mathématique. Certains solveurs commerciaux tels `Ilog CP Optimizer` et `comet` proposent des fonctionnalités supplémentaires concernant le calcul distribué et parallèle. Dans un tout autre registre, le solveur `sugar` utilise une méthode d'*order encoding* [31] transformant un CSP vers un problème de satisfiabilité booléenne (SAT) qui est ensuite résolu efficacement par un des deux solveurs SAT simples `minisat` ou `picosat` Face à la diversité des solveurs et langages, un effort de standardisation des langages de modélisation a été récemment entrepris à l'instar de ce qui existe en programmation mathématique. Cependant, cet effort d'uniformisation des contraintes globales et des langages de description de CSP ne permettent pas encore de déclarer un problème sans aucune considération du solveur sous-jacent. Simultanément, il est encore nécessaire de développer des contraintes globales dédiées à la résolution d'un problème précis.

Au début de cette thèse, mon choix s'est naturellement porté sur le solveur `choco` développé au sein de l'École des Mines de Nantes pour bénéficier de l'expertise de mes collègues lors de mon apprentissage. La lecture et la modification du code source peuvent être un facteur clé de l'apprentissage et la compréhension tout en augmentant l'autonomie des utilisateurs chevronnés. Finalement, la participation à un projet dynamique et collaboratif a enrichi ma réflexion grâce aux rencontres, discussions et collaborations qui s'en sont suivies. Nous reviendrons en détail sur nos contributions au solveur Choco dans le chapitre 9 et en annexes A et B.

À titre d'exemple, le Listing 1 montre la déclaration et la résolution du modèle en figure 2.5 pour le problème des n reines avec l'API du solveur Choco. Une instance d'un CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ est stockée dans un objet de la classe `CPModel` implémentant l'interface `Model`. On déclare d'abord les variables principales et auxiliaires stockées dans trois tableaux grâce aux fonctions de la fabrique `Choco.*` en précisant leurs domaines initiaux et leurs noms. On pose ensuite les contraintes de liaison et les contraintes globales dont les variables sont automatiquement extraites et ajoutées au modèle. On déclare ensuite un objet de la classe `CPSolver` implémentant l'interface `Solver` qui gère les structures de données nécessaires à la résolution. La lecture du modèle construit les objets utiles à la résolution en fonction d'options du modèle et de la structure du problème. Elle détermine entre autres les types des domaines (intervalle, énumération ...) et le filtrage des contraintes (consistance d'arcs, consistance de bornes ...). On lance ensuite l'exploration pour trouver toutes les solutions du problème avec la configuration par défaut (algorithmes de recherche, méthodes de séparation, heuristiques de sélection).

Listing 1 – Code source Java utilisant choco pour calculer toutes les solutions du problème des n reines décrit en figure 2.5.

```
Model m = new CPModel();
IntegerVariable[] queens = Choco.makeIntVarArray("Q", n, 1,n);
IntegerVariable[] p = Choco.makeIntVarArray("p", n, 1, 2*n);
IntegerVariable[] q = Choco.makeIntVarArray("q", n, -n, n);
for (int i = 0; i < n; i++) {
    m.addConstraints(eq(p[i], minus(queens[i], i)), eq(q[i], plus(queens[i], i)));
}
m.addConstraints(allDifferent(queens),allDifferent(p),allDifferent(q));
Solver s = new CPSolver();
s.read(m); s.solveAll();
```

2.6 Conclusion

Nous avons décrit dans ce chapitre les principes de la Programmation Par Contraintes (PPC). Cette approche déclarative permet de résoudre des problèmes combinatoires variés. Les utilisateurs décrivent leur problème en posant des contraintes sur les valeurs que peuvent prendre les différentes variables composant le problème. Un solveur de contraintes est alors chargé de calculer les solutions du problème. Le processus de résolution exacte de problèmes de satisfaction de contraintes permet de générer efficacement les solutions d'un problème grâce à la combinaison des techniques de filtrage et des algorithmes de recherche.

Chapitre 3

Ordonnancement sous contraintes

Nous introduisons les notions fondamentales en ordonnancement sous contraintes : tâches, contraintes temporelles et contraintes de partage de ressource. Nous évoquons brièvement les techniques de placement sous contraintes apparentées à nos travaux en ordonnancement.

Sommaire

3.1 Tâches . . . 26
3.2 Contraintes temporelles . . . 27
3.2.1 Contraintes de précédence . . . 27
3.2.2 Contraintes de disponibilité et d'échéance . . . 27
3.2.3 Contraintes de disjonction . . . 27
3.2.4 Problèmes temporels . . . 28
3.3 Contraintes de partage de ressource . . . 28
3.3.1 Contrainte disjonctive . . . 29
3.3.2 Contrainte cumulative . . . 30
3.4 Modèle disjonctif . . . 31
3.5 Placement sous contraintes . . . 32
3.5.1 Placement en une dimension . . . 32
3.5.2 Placement en deux dimensions . . . 33
3.6 Conclusion . . . 34

Les problèmes d'ordonnancement et de placement forment une classe de problèmes d'optimisation combinatoire généralement complexe. La programmation par contraintes est dorénavant une approche efficace pour ces problèmes.

Nous rappelons dans ce chapitre les notions d'ordonnancement et placement sous contraintes nécessaires à la compréhension de nos travaux en insistant sur la conception de mécanismes efficaces et généraux de propagation de contraintes qui réduisent l'espace de recherche. Le lecteur est invité à se référer à Lopez et Roubellat [32] ou Baptiste *et al.* [33] pour un état de l'art approfondi. Nous introduisons la notion de tâche (section 3.1) puis deux grandes familles de contraintes d'ordonnancement : les contraintes temporelles (section 3.2) et les contraintes de partage de ressource (section 3.3). Nous présentons ensuite le modèle disjonctif (section 3.4) qui sera discuté dans nos travaux sur les problèmes d'atelier. Nous ne traiterons pas des stratégies de recherche dans ce chapitre. Elles seront introduites au cas par cas dans la deuxième partie de la thèse.

Finalement, nous rappelons quelques résultats de placement sous contraintes (section 3.5). En effet, nous utilisons une contrainte globale de placement à une dimension dans nos travaux sur les problèmes de fournées. Par ailleurs, nous insisterons sur la parenté entre les problèmes d'ordonnancement et certains problèmes de placement à deux dimensions.

3.1 Tâches

Une tâche ou activité $i \in [1, n]$, notée T_i, est une entité élémentaire de travail localisée dans le temps par une date de début s_i (*start*) et de fin e_i (*end*), dont la réalisation est caractérisée par une durée positive p_i (*processing time*). La fonction booléenne $\delta_i(t)$ est la fonction de Dirac de l'ensemble des moments $\mathcal{M}_i$ où la tâche T_i est exécutée :

$$s_i = \min(\mathcal{M}_i) \qquad e_i = \max(\mathcal{M}_i) \qquad p_i = \text{card}(\mathcal{M}_i)$$

Une tâche est dite préemptive si elle peut être morcelée afin de diminuer l'inactivité des ressources ($s_i + p_i \leq e_i$). Dans les problèmes non préemptifs, les tâches ne peuvent pas être interrompues ($s_i + p_i = e_i$ et $\mathcal{M}_i = [s_i, e_i[$).
Nous définissons quelques notations relatives à l'exécution d'une tâche T_i en suivant la terminologie anglaise de la littérature illustrée en figure 3.1. Les notations $est_i = \min(s_i)$ et $lst_i = \max(s_i)$ désigneront respectivement les dates de début au plus tôt (*earliest starting time*) et au plus tard (*latest starting time*) d'une tâche T_i alors que $ect_i = \min(e_i)$ et $lct_i = \max(e_i)$ désigneront ses dates de fin au plus tôt (*earliest completion time*) et au plus tard (*latest completion time*). La fenêtre de temps d'une tâche (*time window*) désigne l'intervalle $[est_i, lct_i[$ durant lequel la tâche est potentiellement exécutée. La partie obligatoire (*compulsory part*) [34] d'une tâche désigne l'intervalle $[lst_i, ect_i[$ durant lequel la tâche est obligatoirement exécutée (représenté par un rectangle dans la figure 3.1). Notez que la partie obligatoire d'une tâche peut être vide.

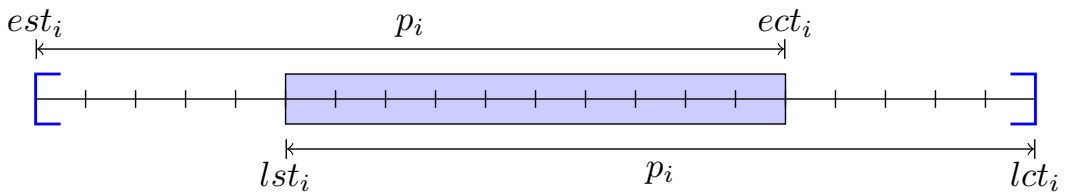


FIGURE 3.1 – Notations relatives à l'exécution d'une tâche.

Deux tâches fictives de durées nulles, T_{start} et T_{end}, représentant le début et la fin de l'ordonnancement sont généralement ajoutées. Elles sont généralement reliées aux autres tâches par les contraintes arithmétiques $s_i - s_{\text{start}} \geq 0$ et $e_{\text{end}} - e_i \geq 0$. Sans perte de généralité, nous supposons dorénavant que le début de l'ordonnancement est fixé à l'origine des temps ($s_{\text{start}} = 0$) et que la fin du projet est égale à son délai total ($e_{\text{end}} = \max_I(e_i)$).

Pour une tâche non préemptive, ou non morcelable, l'ensemble des moments d'exécution $\mathcal{M}_i$ est un intervalle. Une unique variable s_i ou e_i suffit à caractériser une tâche non préemptive de durée fixe. Un couple de variables (s_i, p_i), (s_i, e_i) ou (p_i, e_i) suffit à caractériser une tâche non préemptive de durée variable, c'est-à-dire qui n'est pas connue à l'avance. Néanmoins, `choco` utilise un triplet de variables entières positives (s_i, p_i, e_i) sous une contrainte de non-préemption $s_i + p_i = e_i$. La raison principale est la granularité plus fine offerte par cette représentation pour les tâches à durée variable. Par exemple, une partie de l'information sur la tâche ($s_i \in [0, 10], p_i \in [1, 5], e_i \in [1, 11]$) est nécessairement perdue lorsqu'elle est seulement représentée par un couple de variables. Ce choix est aussi guidé par des raisons plus spécifiques de `choco` détaillées dans le chapitre 9. Les expérimentations suggèrent que ce choix a une influence acceptable sur les performances du solveur lorsque toutes les tâches ont des durées fixes.

L'objectif des problèmes d'ordonnancement est de minimiser une mesure de performance. Certaines dépendent exclusivement des dates de fin des tâches, notée C_i selon les conventions de recherche opérationnelle. Le délai total ou date d'achèvement maximale d'un ordonnancement C_{max} (*makespan*) est la date de fin de la dernière tâche à quitter le système. Une valeur minimale de C_{max} correspond généralement à une utilisation intensive des ressources. La délai moyen, pondérée $\sum w_i C_i$ ou non $\sum C_i$ peut aussi indiquer le coût de l'ordonnancement.

3.2 Contraintes temporelles

Cette section traite de la représentation des contraintes temporelles en ordonnancement sous contraintes. Elles intègrent les contraintes de temps alloué issues généralement d'impératifs de gestion et relatives aux dates limites des tâches (disponibilité des approvisionnements, délai de livraison) ou au délai total d'un projet, mais aussi les contraintes d'enchaînement qui définissent le positionnement relatif de certaines tâches par rapport à d'autres.
Ces contraintes peuvent toutes s'exprimer à l'aide d'*inégalités de potentiels* [35] qui imposent une distance minimale d_{ij} entre deux instants particuliers associés aux tâches (le plus souvent les dates de début) : $x_j - x_i \geq d_{ij}$. Ainsi, le réseau de contraintes est souvent représenté par un graphe de distance $\mathcal{G} = (\mathcal{N}, V)$ dans lequel les nœuds représentent les variables et les arcs (i, j) de valuation d_{ij} représentent les contraintes temporelles. Les deux tâches fictives T_{start} et T_{end} sont généralement ajoutées à ce graphe par les contraintes arithmétiques présentées précédemment.
Dans cette thèse, nous n'utiliserons que les contraintes temporelles introduites dans cette section.

3.2.1 Contraintes de précédence

Une contrainte de précédence entre deux tâches T_i et T_j, notée symboliquement $T_i \prec T_j$ (T_i précède T_j), est représentable par une unique inégalité de potentiel (3.1). Il s'agit d'un cas particulier de la contrainte de précédence avec temps d'attente $d_{ij} \geq 0$ (3.2). Le temps d'attente représente, par exemple, le temps d'entretien d'une machine, le temps de refroidissement ou de séchage d'une pièce, etc.

$$T_i \preceq T_j \quad \Leftrightarrow \quad s_j - e_i \geq 0 \tag{3.1}$$
$$T_i \prec T_j \quad \Leftrightarrow \quad s_j - e_i \geq d_{ij} \tag{3.2}$$

Le filtrage d'une précédence consiste généralement à appliquer une règle d'arc-consistance pour l'inégalité de potentiel sur les bornes des variables.

3.2.2 Contraintes de disponibilité et d'échéance

Les contraintes de dates limites d'une tâche T_i peuvent également s'exprimer à l'aide d'inégalités de potentiel entre T_i, T_{start} et T_{end}. La contrainte de disponibilité $s_i \geq r_i$ se réécrit sous la forme $s_i - s_{\text{start}} \geq r_i$ et la contrainte d'échéance $e_i \leq d_i$ devient $s_{\text{start}} - e_i \geq -d_i$. En général, on réduit directement les fenêtres de temps (via les domaines) lorsque les dates limites sont connues à l'avance.

3.2.3 Contraintes de disjonction

Une expression peut relier plusieurs inégalités de potentiel par des connecteurs logiques. Une disjonction d'inégalités de potentiel est une expression dans laquelle apparaît le connecteur logique $\vee$ (ou logique). Sur le plan sémantique, la contrainte est satisfaite si au moins un des littéraux est une contrainte satisfaite. Une contrainte de disjonction, ou non-chevauchement, entre deux tâches T_i et T_j est satisfaite si les tâches s'exécutent dans des fenêtres de temps disjointes (3.3). Il s'agit à nouveau d'un cas particulier de la contrainte de disjonction avec temps d'attente (3.4). Cette contrainte peut par exemple exprimer l'existence de deux gammes opératoires différentes. Arbitrer une disjonction consiste à déterminer l'ordre relatif entre les tâches, c'est-à-dire choisir quelle précédence (littéral) est satisfaite.

$$T_i \simeq T_j \quad \Leftrightarrow \quad (T_i \preceq T_j) \vee (T_j \preceq T_i) \tag{3.3}$$
$$T_i \sim T_j \quad \Leftrightarrow \quad (T_i \prec T_j) \vee (T_j \prec T_i) \tag{3.4}$$

Tant que la disjonction n'est pas arbitrée, le filtrage applique deux règles de séquencement : Précédence Interdite (PI) - Précédence Obligatoire (PO). Une précédence $T_i \prec T_j$ est dite interdite lorsque les fenêtres de temps des tâches sont incompatibles avec la décision, ce qui rend l'autre précédence obligatoire. Une précédence $T_i \prec T_j$ est dite obligatoire dès qu'elle est directement impliquée par les fenêtres de temps. La règle (PI) implique (PO), mais elles réagissent à différents types de modification des domaines. Ces

règles détectent une inconsistance lorsqu'aucun séquencement n'est possible.

$$ect_i + d_{ij} \geq lst_j \quad \Rightarrow \quad T_j \prec T_i \tag{PI}$$
$$lct_i + d_{ij} \leq est_j \quad \Rightarrow \quad T_i \prec T_j \tag{PO}$$

3.2.4 Problèmes temporels

Les problèmes temporels consistent à ordonnancer un ensemble de tâches liées uniquement par des contraintes temporelles [36]. Le *problème d'ordonnancement de projet* connu sous l'acronyme PERT consiste à ordonnancer un ensemble de tâches liées par des contraintes de précédence sans limitation de ressources. Le *problème central de l'ordonnancement* consiste à ordonnancer un ensemble des tâches liées par des contraintes de précédences généralisées (le temps d'attente peut être négatif). Nous supposons que $d_{ij} \geq -d_{ji}$ pour éviter que le problème ne soit trivialement insatisfiable. Le graphe de distance permet de résoudre ce problème grâce à sa transformation en un problème (polynomial) de flot ou de plus court chemin [37]. De plus, il a été montré que le problème est consistant en l'absence de cycle de coût négatif et peut, dans ce cas, être résolu sans backtrack. Il est donc possible de calculer les ordonnancements au plus tôt et au plus tard en temps polynomial et même linéaire dans le cas du problème d'ordonnancement de projet. Les ordonnancements au plus tôt et au plus tard peuvent alors servir à la réduction de la fenêtre des temps des tâches même en présence de limitation de ressources. Nous reviendrons sur la résolution du problème d'ordonnancement de projet avec le solveur Choco en Annexe B. Quand on ne peut pas résoudre le problème sans backtrack, certains travaux portent sur la détection de cycle et la propagation incrémentale lors de l'ajout ou du retrait dynamique de contraintes temporelles.

3.3 Contraintes de partage de ressource

Une ressource r est un moyen technique ou humain requis pour la réalisation d'une tâche et disponible en quantité limitée, sa capacité entière positive C_r (supposée constante). Plusieurs types de ressource sont à distinguer. Une ressource est *renouvelable* si après avoir été utilisée par une ou plusieurs tâches, elle est à nouveau disponible en même quantité (les humains, les machines, l'espace ...). Dans le cas contraire, elle est *consommable* (matières premières, budget ...), c'est-à-dire la consommation globale au cours du temps est limitée. Une ressource peut aussi être *doublement contrainte* lorsque son utilisation instantanée et sa consommation globale sont toutes deux limitées (source d'énergie, financement ...). D'autres types de ressource (producteur – consommateur, taux de renouvellement ...) définis en fonction de l'évolution de la capacité instantanée ou globale lors de l'exécution des tâches (produits manufacturés ou naturels ...) ne seront pas discutés dans cette thèse. On distingue par ailleurs les ressources disjonctives ($C_r = 1$) qui ne peuvent exécuter qu'une tâche à la fois, des ressources cumulatives ($C_r \geq 1$) qui peuvent être utilisées par plusieurs tâches simultanément.
La consommation d'une tâche T_i sur une ressource renouvelable r est caractérisée par une consommation totale d'énergie $w_i(r) \geq 0$, une consommation minimale à chaque instant ou hauteur $h_i(r)$ et une consommation instantanée $w_i(r,t)$ à l'instant t. Dans le cas *fully elastic* (FE), la capacité de la ressource doit être respectée à chaque instant, mais il n'y a aucune restriction sur l'utilisation de la ressource par les tâches. La contrainte (3.5) impose que l'énergie consommée par une tâche lors de son exécution soit supérieure à sa consommation minimale requise. La contrainte (3.7) limite l'énergie consommée par une tâche à chaque instant. La contrainte (3.6) assure que l'énergie totale consommée par une tâche sur l'horizon de planification $[0, h]$ est égale à sa consommation totale. Finalement, la contrainte (3.8) limite l'énergie totale consommée par les tâches à chaque instant. Les contraintes supplémentaires $w_i(r,t) = \delta_i(t)$ et $w_i(r,t) = h_i(r) \times \delta_i(t)$ sont respectivement ajoutées pour représenter la consommation constante des tâches sur les ressources disjonctives et cumulatives. Soit $u_i(r)$ une variable booléenne indiquant si la tâche i utilise la ressource r, le modèle obtenu en substituant l'expression non linéaire $\delta_i(t) \times u_i(r)$ à la variable $\delta_i(t)$ est valable pour les problèmes d'allocation de ressources.

$$w_i(r,t) \geq h_i(r) \times \delta_i(t) \quad (3.5) \qquad w_i(r,t) \leq C_r \times \delta_i(t) \quad (3.7)$$

$$\sum_{t=0}^{h} w_i(r,t) = w_i(r) \quad (3.6) \qquad \sum_{i=1}^{n} w_i(r,t) \leq C_r \quad (3.8)$$

Cette représentation induit une hiérarchie entre les ressources illustrée par le diagramme de la figure 3.2 dans lequel la relation A → B indique que le type de ressource B est un cas particulier du type A. Le cas

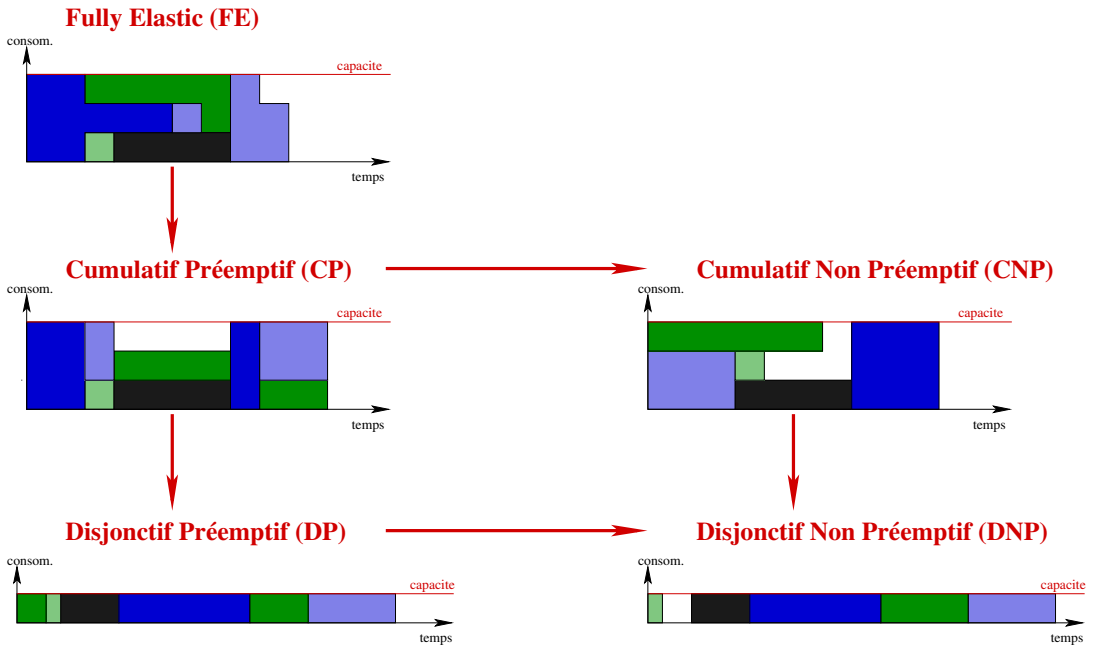


FIGURE 3.2 – Hiérarchie des types de ressources.

général *fully elastic* est intéressant puisque ses propriétés et règles de filtrage sont valides pour tous les autres types de ressource. Les techniques de *time-tabling* permettent de réaliser le filtrage des ressources en maintenant l'arc-consistance sur les contraintes (3.8) puis l'ajustement des dates de début et de fin des tâches en fonction de leur consommation instantanée d'énergie $w_i(r,t)$. Elles sont peu efficaces dans le cas général, souvent appliquées dans le cas préemptif, et spécialisées dans le cas non préemptif. Nous présentons maintenant les principes du filtrage des contraintes disjonctive et cumulative dans le cas non préemptif.

3.3.1 Contrainte disjonctive

La prise en compte d'une contrainte disjonctive définit un problème de séquencement dans lequel il faut ordonner totalement les tâches qui utilisent la ressource. La prise en compte des fenêtres de temps réduit l'ensemble des solutions : certaines configurations sont interdites ; d'autres deviennent obligatoires.

Le concept d'énergie permet d'effectuer des raisonnements quantitatifs intégrant les contraintes de temps et de ressource. Ces raisonnements font appel à un bilan de l'utilisation d'une ressource sur un nombre quadratique d'intervalles temporels pertinents [38] pour l'identification et le calcul de différentes énergies. Sur un intervalle de temps, l'énergie est fournie par la ressource et consommée par les tâches. La règle d'*overload checking* (OC) consiste à lever une contradiction lorsque le bilan énergétique d'un intervalle temporel est déficitaire, c'est-à-dire la consommation d'énergie minimale des tâches sur cet intervalle dépasse l'énergie totale disponible. Ce type de raisonnement permet d'interdire la localisation des tâches dans certains intervalles temporels qui engendrerait un bilan énergétique déficitaire.

D'autres règles de filtrage dominant les raisonnements énergétiques dans le cas disjonctif déduisent des conditions de séquencement des tâches qui sont ensuite propagées et peuvent entraîner un ajustement [39]

des fenêtres d'exécution des tâches.
La règle *not first/not last* (NF/NL) généralise la règle (PI) en déduisant qu'une tâche T_i ne peut être ordonnancée avant/après un sous-ensemble Ω d'autres tâches partageant la ressource disjonctive. Cette règle entraîne un ajustement de la fenêtre de temps, car il existe au moins une tâche de Ω placée avant/après la tâche T_i dans un ordonnancement admissible. Baptiste et Le Pape [38], Torres et Lopez [40] ont proposé des algorithmes pour cette règle dont la complexité est quadratique.
Réciproquement, la règle d'*edge finding* (EF) détermine qu'une tâche T_i est obligatoirement ordonnancée avant/après un sous-ensemble de tâches Ω. La fenêtre de temps de la tâche est alors ajustée en estimant la date de fin au plus tôt ou de début au plus tard des tâches de l'ensemble Ω. Caseau et Laburthe [41, 42] ont proposé un algorithme en $O(n^3)$ basé sur les intervalles de tâches. Ils maintiennent ces intervalles de tâches incrémentalement durant la recherche et utilisent un système de règle d'activation des intervalles pour améliorer l'efficacité de leur méthode. Des déductions supplémentaires peuvent être réalisées grâce aux intervalles de tâches. Carlier et Pinson [39] ont proposé le premier algorithme en $O(n \log n)$ n'utilisant pas les intervalles de tâches mais une structures de données plus complexe.

Vilím [43], Vilím *et al.* [44] ont récemment proposé une implémentation des règles précédentes avec une complexité en $O(n \log n)$. Ils utilisent une structure d'arbre binaire équilibré qui leur permet d'abaisser la complexité et de proposer une nouvelle règle de filtrage : *detectable precedence* (DP). Cette règle propage les précédences découvertes en inspectant les fenêtres de temps des tâches. La fenêtre de temps de chaque tâche est ajustée en fonction de l'ensemble de ses prédécesseurs ou successeurs. Cette règle s'applique quelquefois lorsque les règles précédentes ne permettent aucun ajustement. En effet, l'*edge finding* peut identifier que T_i précède T_k et que T_j précède T_k ($T_i \preceq T_k$ et $T_j \preceq T_k$). Il peut arriver qu'aucune de ces précédences ne permette d'ajustement des bornes. L'idée consiste alors à exploiter l'information $\{T_i, T_j\} \preceq T_k$.
Tous les algorithmes reposent sur le calcul de la date de fin au plus tôt ECT_Ω (*earliest completion time*) d'un ensemble de tâches Ω partageant une ressource disjonctive :

$$ECT_\Omega = \max \left\{ \min_{T_i \in \Omega} \{est_i\} + \sum_{T_i \in \Omega'} p_i,\ \Omega' \subseteq \Omega \right\} \tag{3.9}$$

Vilím [43] a proposé des structures de données efficaces : les arbres Θ et Θ-Λ. Ils sont basés sur une structure d'arbre binaire équilibré permettant un calcul incrémental de ECT_Ω avec une complexité de $O(\log(n))$ lors de l'ajout ou du retrait d'une tâche de l'ensemble Ω.

Van Hentenryck *et al.* [45] utilisent la construction de disjonction pour supprimer les valeurs inconsistantes dans les deux séquencements possibles d'une paire de tâches. Plus récemment, Wolf [46] ajuste les fenêtres de temps des tâches en fonction de leur nombre maximal de prédécesseurs et de successeurs. Hooker [47] a proposé différentes relaxations linéaires. La règle de Jackson [48] construit un ordonnancement préemptif qui est une condition nécessaire pour la contrainte disjonctive.

De nombreuses extensions de la contrainte ont été proposées. Par exemple, une version *soft* de la contrainte, c'est-à-dire une relaxation, a été proposée pour maximiser le nombre de tâches exécutées par la ressource sous des contraintes de dates limites [49]. Ce principe a été étendu depuis grâce à la notion de *ressource alternative* pour laquelle certaines tâches sont optionnelles, c'est-à-dire qu'elles peuvent ne pas être exécutées par la ressource [44, 50–52]. Des travaux préliminaires sur un problème d'allocation de ressource seront discutés en Annexe D. Les algorithmes de filtrage dépendent de l'appartenance des tâches non exécutées à l'ordonnancement final ou non.
Barták et Cepek [53] maintiennent aussi la fermeture transitive des précédences concernant les tâches de la ressource pour déduire de nouvelles précédences et ajuster les fenêtres de temps.

3.3.2 Contrainte cumulative

La contrainte cumulative impose que la somme des hauteurs des tâches s'exécutant à un instant donné soit inférieure à la capacité de la ressource. La contrainte peut aussi imposer un niveau de consommation minimale aux instants où des tâches s'exécutent sur la ressource.

Les premiers algorithmes de filtrage sont basés sur la construction d'un profil cumulatif qui agrège les parties obligatoires des tâches afin d'ajuster leurs fenêtres de temps pour éviter un dépassement de la capacité de la ressource. Même si les tâches avec une partie obligatoire vide sont ignorées, l'algorithme à

balayage (*sweep*) proposé par Beldiceanu et Carlsson [54] gèrent des cas généraux (ressources alternatives, producteur – consommateur ...) tout en permettant un passage à l'échelle grâce à une complexité réduite de $O(n \log n)$. Nous rappelons brièvement le principe des algorithmes à balayage : l'algorithme déplace une ligne verticale (*sweep-line*) et utilise principalement deux structures de données :
Sweep-line status : cette structure contient les informations sur la position courante Δ de la *sweep-line*.
Event-point series : cette structure contient les événements à examiner triés en ordre croissant.
L'algorithme initialise la *sweep-line status* à la valeur initiale de Δ. La *sweep-line* parcours ensuite les événements en mettant à jour la *sweep-line status*. Dans notre cas, la *sweep-line* parcourt les valeurs du domaine d'une variable T_i et la *sweep-line status* contient un ensemble de contraintes qui doivent être vérifiées à l'instant Δ. Si pour un instant Δ, l'ensemble des contraintes est insatisfiable, alors on peut enlever la valeur Δ du domaine de T_i.
Une propriété intéressante de cet algorithme est que l'on ne considère qu'un nombre linéaire d'évènements à chaque appel.

Les algorithmes basés sur les raisonnements énergétiques considèrent la consommation d'ensemble de tâches sur certains intervalles temporels sans se restreindre aux parties obligatoires [55]. Ainsi, les règles d'*edge finding* [56] ont été adaptées au cas cumulatif ainsi que l'arbre-Θ-Λ devenu arbre-Φ [57].

Une condition nécessaire pour la contrainte cumulative est obtenue en imposant une contrainte disjonctive sur l'ensemble des tâches tel que la somme des hauteurs de chaque paire de tâches est supérieure à la capacité. Il existe des techniques plus sophistiquées pour extraire les cliques de disjonctions lorsque plusieurs ressources disjonctives et cumulatives sont impliquées [58].

De nombreuses extensions existent : la cumulative colorée où la consommation est égale au nombre de couleurs distinctes associées aux tâches s'exécutant à un instant donné ; la cumulative avec niveau de priorité où une capacité relative à un niveau donné s'applique à l'ensemble des tâches de priorité inférieure à ce même niveau ; les variantes avec des tâches optionnelles.

3.4 Modèle disjonctif

Le modèle disjonctif [59] est un graphe permettant de décrire les contraintes de précédence et de disjonction dans un problème d'ordonnancement utilisé aussi bien par les méthodes approchées que les méthodes exactes, notamment dans les chapitres 6 et 7. Plus précisément, pour un problème d'ordonnancement de n tâches, le *graphe disjonctif généralisé* $\mathcal{G} = (\mathcal{T}, \mathcal{P}, \mathcal{D}, \mathcal{L})$ est défini comme suit :

- $\mathcal{T}$ est l'ensemble des *nœuds*, représentant les tâches ainsi que deux tâches fictives T_{start} et T_{end} ;
- $\mathcal{P}$ est l'ensemble des *arcs* (orientés) représentant les contraintes d'enchaînement ;
- $\mathcal{D}$ est l'ensemble des *arêtes* (non orientées) représentant les disjonctions ;
- $\mathcal{L}$ est l'ensemble des *valuations* positives donnant pour chaque arc (T_i, T_j) le temps d'attente entre la fin de la tâche T_i et le début de la tâche T_j.

Nous l'appellerons simplement *graphe disjonctif* lorsque les valuations $\mathcal{L}$ sont nulles. Le *graphe de précédence* le sous-graphe orienté $(\mathcal{T}, \mathcal{P})$.

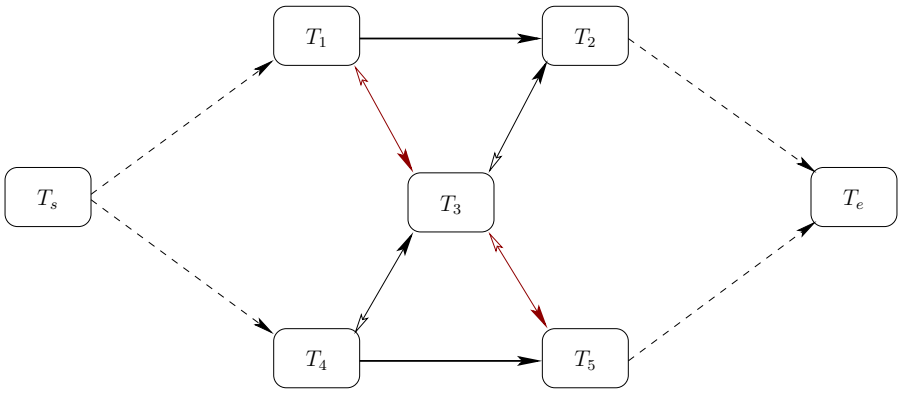
Arbitrer une disjonction revient à transformer une arête $T_i \sim T_j$ en un arc $T_i \prec T_j$ ou $T_j \prec T_i$. La contrainte de non-préemption $s_i + p_i = e_i$ doit aussi être satisfaite en chaque nœud.
L'existence d'un cycle entraîne une inconsistance triviale du problème. De plus, tout ordonnancement réalisable correspond à un arbitrage complet de $\mathcal{G}$, dans lequel toutes les disjonctions sont arbitrées sans créer de circuit. Lorsque toutes les contraintes portant sur les tâches sont représentées dans $\mathcal{G}$, la consistance d'un arbitrage complet implique l'existence d'un ordonnancement dont le délai total correspond à la longueur d'un plus long chemin de $\mathcal{G}$ appelé *chemin critique*. Les activités de ces chemins (il peut en exister plusieurs), dites activités critiques, déterminent à elles seules la date de fin du projet C_{max}. Il est nécessaire de réduire la longueur des chemins critiques pour réduire C_{max}. La marge (*slack time*) est le temps pendant lequel une tâche peut être retardée sans retarder le projet. Le temps de latence des activités critiques est donc nul alors que celui des activités non critiques est strictement positif. Pour un graphe de k arcs, ce chemin est calculable en $O(k)$ par une variante de l'algorithme de Bellman [37].
Beaucoup de métaheuristiques et de recherches arborescentes explorent les arbitrages complets pour en trouver un de délai total minimum. Dans une recherche arborescente, une *heuristique de sélection de disjonction* sélectionne la prochaine disjonction à arbitrer, alors qu'une *heuristique de sélection d'arbitrage* détermine la première précédence à essayer.

T_i	1	2	3	4	5
p_i	6	7	5	11	2
est_i	0	16	11	0	16
lst_i	5	16	11	21	16

(a) Durées d'exécution

$$T_1 \preceq T_2 \quad T_4 \preceq T_5$$
$$T_1 \simeq T_3 \quad T_2 \simeq T_3 \quad T_4 \simeq T_3 \quad T_5 \simeq T_3$$

(b) Contraintes temporelles



(c) Arbitrage d'un graphe disjonctif

FIGURE 3.3 – Construction et d'arbitrage d'un graphe disjonctif.

La construction et l'arbitrage complet d'un graphe disjonctif sont illustrés dans la figure 3.3. Nous cherchons à minimiser le délai total de l'ordonnancement des tâches définies dans la figure 3.3(a) sous les contraintes temporelles en figure 3.3(b). L'arbitrage du graphe disjonctif de la figure 3.3(c) représente des ordonnancements optimaux de délai total égal à 23. Un sommet du graphe disjonctif est associé à chaque tâche et les arcs orientés représentent les précédences alors que les arcs bidirectionnels représentent les disjonctions dont les arbitrages sont indiqués par la direction de la flèche pleine. Le chemin $(T_{\text{start}}, T_4, T_3, T_2, T_{\text{end}})$ de longueur $11+5+7=23$ est l'unique chemin critique. Les valeurs est_i et lst_i de la figure 3.3(a) correspondent aux ordonnancements au plus tôt et au plus tard de cet arbitrage complet.

3.5 Placement sous contraintes

Les problèmes de placement consistent à ranger des articles caractérisés par leur forme(s) dans une ou plusieurs boîtes. Les variantes se distinguent en fonction de la dimension, de la connaissance a priori des articles (*on-line*, ou *off-line*), de la forme des articles et des boîtes (carré, rectangulaire, circulaire ...), de la possibilité de modifier l'orientation des articles. On distingue aussi le problème de faisabilité, c'est-à-dire, existe-t-il un rangement réalisable des articles dans les boîtes, des problèmes d'optimisation, par exemple la minimisation du nombre de boîtes (*bin packing*) ou la minimisation des dimensions d'une seule boîte (hauteur – *strip packing*, aire – *rectangle packing*, volume ...), ou encore la maximisation de la valeur du rangement (problèmes de sac à dos – *knapsack problems*). Le lecteur peut se référer à la bibliographie annotée de [60] ou à une typologie précise des problèmes de placement et de découpe [61, 62]. Dans cette section, nous nous intéresserons particulièrement aux problèmes de placement à une ou deux dimensions qui partagent certaines propriétés avec les problèmes d'ordonnancement.

3.5.1 Placement en une dimension

Nous considérons dans cette section un ensemble d'articles I caractérisés par leur longueur entière positive s_i et un ensemble de conteneurs caractérisés par leur capacité C ($s_i \leq C$).
Le problème *subset sum* n'est pas à proprement parler un problème de placement ou d'ordonnancement, mais il apparaît parfois comme un sous-problème pour obtenir des évaluations par défaut lors du filtrage. Il consiste à maximiser l'espace occupé par le rangement d'un sous-ensemble d'articles dans une boîte. Ce problème est NP-complet malgré son apparente simplicité. Toutefois, il est résolu très efficacement

par la programmation dynamique avec une complexité pseudo linéaire $O(\max(s_i)n)$ [63].
Le problème de *bin packing* à une dimension consiste à minimiser le nombre de boîtes nécessaires pour ranger un ensemble d'articles. De nombreux travaux traitent de la qualité des solutions fournies par des algorithmes polynomiaux d'approximation. Par exemple, les heuristiques de permutation *first fit decreasing* ou *best fit decreasing* rangent les articles par ordre décroissant de taille en les plaçant respectivement dans le premier conteneur disponible ou dans un conteneur disponible dont l'espace restant est minimal. Une permutation des articles définit une classe de rangement.
La recherche arborescente en profondeur de Martello et Toth [64] considère en chaque nœud le rangement du plus grand article restant dans une boîte partiellement remplie ou vide. Chaque nœud est évalué par le calcul d'une borne inférieure dédiée et de la meilleure borne supérieure fournie par des heuristiques. Depuis, diverses approches exactes ont été proposées basées sur des formulations en nombres entiers couplées avec des relaxations linéaires, notamment par la génération de colonnes [65], des méthodes hybrides [66] ou la programmation par contraintes [67–72]. Korf [69], Fukunaga et Korf [71] utilisent un schéma de séparation *bin completion* différent de celui de Martello et Toth [64] qui consiste à ajouter une boîte réalisable au rangement à chaque nœud. Une boite réalisable contient un sous-ensemble des articles libres respectant la capacité de la boîte. Toutes les méthodes mentionnées ci-dessus utilisent intensivement des règles d'équivalence et de dominance ainsi que des bornes inférieures et supérieures dynamiques pour réduire l'espace de recherche. Par exemple, l'efficacité de la *bin completion* dépend exclusivement des règles de dominances entre les boîtes réalisables.
Au contraire, Shaw [70] a proposé une contrainte globale dont le filtrage associé au modèle de Martello et Toth [64] utilisent des raisonnements de type sac à dos et une borne inférieure sur le nombre de boîtes exploitant l'affectation partielle courante indépendamment de toute symétrie ou dominance. La signature de la contrainte contient deux ensembles de variables représentant respectivement la boîte où un article est rangé et l'ensemble des articles rangés dans une boîte. La seule restriction concerne la longueur des articles qui doit être connue à l'avance, mais les autres variables ont des domaines arbitraires. Ainsi, sa conception favorise son intégration dans des modèles complexes avec des contraintes additionnelles (incompatibilités, précédences) ou des objectifs différents tout en restant efficace sur les purs problèmes de *bin packing*. Par exemple, Schaus et Deville [72] ont complété le filtrage pour la gestion de contraintes de précédences dans des problèmes de chaîne d'assemblage. Cependant, le filtrage des rangements réalisables dans une seule boîte n'atteint pas la consistance d'arc généralisée au contraire de celui proposé par [73] pour les problèmes de sac à dos. Nous utiliserons cette contrainte globale dans le modèle pour le problème d'ordonnancement d'une machine à traitement par fournées au chapitre 8.

3.5.2 Placement en deux dimensions

Nous considérons un ensemble d'articles I de forme rectangulaire caractérisés par leur largeur w_i et leur hauteur h_i et de conteneurs caractérisés par leur largeur W et leur hauteur H. Le rangement des articles dans un conteneur respecte les trois contraintes suivantes : (a) chaque article est entièrement inclus dans le conteneur ; (b) Les articles ne se chevauchent pas ; (c) Le rangement est orthogonal, c'est-à-dire le bord des articles est parallèle au bord du conteneur. Dans la plupart des travaux mentionnés, l'orientation des articles est fixée. Nous nous restreindrons aux approches en programmation par contraintes. Le lecteur intéressé pourra se référer à la classification proposée par Dyckhoff [61] et à l'état de l'art de Lodi *et al.* [74] sur les modèles mathématiques, les bornes inférieures, les méthodes d'approximation et les heuristiques et métaheuristiques.

Plusieurs modèles coexistent pour représenter une solution au problème de faisabilité qui consiste à trouver un rangement admissible des articles dans une boîte. Dans le plus intuitif, une solution est un vecteur contenant les coordonnées de chaque article. Fekete et Schepers [75], Fekete *et al.* [76] ont proposé un modèle basé sur un graphe d'intervalle pour chaque dimension (un graphe d'intervalle est le graphe d'intersection d'un ensemble d'intervalles). Ceux-ci correspondent à une *classe de rangement*, c'est-à-dire un ensemble de rangements partageant certaines propriétés. Cette représentation évite l'énumération d'un grand nombre de rangements symétriques appartenant à la même classe. Les algorithmes pour reconnaître une classe de rangement réalisable et reconstruire une solution sont polynomiaux. Moffitt et Pollack [77] utilisent le concept de *meta-CSP* dans lequel deux graphes représentent les disjonctions sur les deux dimensions. Ce concept peut être vu comme une extension du modèle disjonctif dans le cas à

deux dimensions. La différence principale réside dans le fait que l'arbitrage d'une disjonction entre deux articles est réalisé sur l'une ou l'autre dimension. Korf [78] utilise simplement une matrice booléenne binaire de la taille du conteneur dans laquelle les cellules de la matrice représentent une coordonnée dont la valeur est 1 si elle est occupée par un rectangle. Le dernier modèle est surtout répandu parmi les heuristiques et métaheuristiques, mais il est intéressant de par sa simplicité : une permutation des articles définit une classe de rangement. Par exemple, l'heuristique *bottom-left* [79] consiste à ranger à chaque itération le prochain article de la liste à la position le plus en bas et à gauche de la boîte.

Moffitt et Pollack [77] cherchent à trouver un conteneur d'aire minimale contenant tous les articles grâce au *meta-CSP*. On peut remarquer le lien entre cet objectif et la minimisation du délai total dans le modèle disjonctif. Cependant, son approche peut être facilement adaptée au problème de faisabilité. Il emploie différentes techniques pour réduire son espace de recherche, notamment le *forward-checking*, l'utilisation de la transitivité pour éliminer des variables, ainsi que le *semantic branching* qui consiste à utiliser l'espace déjà exploré pour rajouter des *coupes*. Sa stratégie de branchement détecte et élimine certaines symétries des placements, et utilise des heuristiques de sélection de variable et de valeur.
Korf [78, 80] traite le même problème par l'application de règles de dominance entre affectations partielles et d'une borne inférieure sur l'aire perdue dans le conteneur sur une représentation matricielle.
D'autres approches sont basées sur l'ordonnancement cumulatif. Le problème d"ordonnancement cumulatif est une relaxation du problème de faisabilité consistant à relâcher la contrainte d'intégrité des rectangles. La satisfiabilité du problème cumulatif associé à chaque dimension d'un problème de faisabilité est même une condition nécessaire à la satisfiabilité du problème de faisabilité mais pas une condition suffisante. Nous illustrons ce point sur un exemple issu de [12] présenté en figure 3.4.

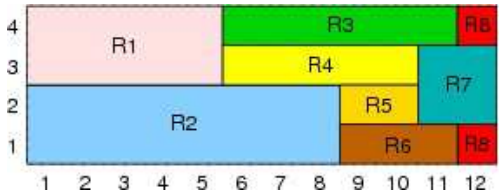


FIGURE 3.4 – Une condition nécessaire mais pas suffisante pour le problème de faisabilité.

Clautiaux *et al.* [81] utilisent une contrainte cumulative sur chaque dimension et vérifie les contraintes de non-chevauchement grâce à un modèle basique. Le filtrage est renforcé par (a) des raisonnements énergétiques adaptés au cas bidimensionnel, (b) une estimation de l'aire inoccupée dans le conteneur par la résolution de plusieurs problèmes *subset sum*, (c) le calcul de bornes inférieures basées sur les *data-dependent dual-feasible functions* [82], (d) l'élimination de certaines symétries de blocs présentées dans Scheithauer [83]. Cette méthode présente l'avantage de détecter efficacement les instances insatisfiables.
Beldiceanu et Carlsson [84] utilisent une contrainte globale de non-chevauchement qui est le cas bidimensionnel de la contrainte `diffn` [85] et assure donc le respect des contraintes de non-chevauchement entre articles. La contrainte utilise un algorithme à balayage basée sur la notion de régions interdites qui représente l'ensemble des origines du rectangle i telles qu'il chevauche obligatoirement le rectangle j. À chaque appel, l'algorithme de filtrage vérifie qu'il existe une position n'appartenant à aucune région interdite pour chaque borne d'un article. Dans le cas contraire, on met à jour la borne concernée grâce à l'algorithme de balayage. Beldiceanu *et al.* [86] ont étendu l'utilisation des algorithmes à balayage pour le traitement de problèmes placement complexes (multidimensionnel, polymorphisme ...) en combinaison avec d'autres algorithmes de filtrage (cumulatif, estimation de l'espace inoccupée ...).

3.6 Conclusion

Ainsi, les problèmes d'ordonnancement et de placement sont intensivement étudiés par les chercheurs en programmation par contraintes. La diversité des approches proposées permet de traiter de nombreux problèmes mais demande une expertise certaine pour déterminer les modèles et techniques appropriées à la résolution d'un problème particulier. Nous verrons dans la seconde partie de cette thèse qu'il en va de même pour les stratégies de recherche.

Chapitre 4

Ordonnancement d'atelier

Nous rappelons la définition et la classification des problèmes d'atelier ainsi que les principaux résultats de complexité. Nous établissons ensuite un état de l'art en insistant sur le problème d'atelier à cheminement libre (open-shop), le problème de base retenu lors de nos expérimentations.

Sommaire

4.1 Définition des problèmes d'atelier . . . 35
4.1.1 Définition des problèmes de base . . . 36
4.1.2 Variantes et classification de Lawler . . . 36
4.1.3 Applications . . . 37
4.2 Résultats de complexité . . . 38
4.2.1 Problèmes à deux machines . . . 38
4.2.2 Problèmes à trois machines : la frontière . . . 38
4.2.3 Cas général . . . 38
4.2.4 Conclusion . . . 39
4.3 État de l'art . . . 39
4.3.1 Jeux d'instances . . . 39
4.3.2 Méthodes approchées . . . 39
4.3.3 Méthodes Exactes . . . 41
4.4 Orientation de nos travaux . . . 42

L'ordonnancement d'atelier consiste à exploiter au mieux des moyens limités, les machines, pour réaliser un ensemble varié de produits, les lots. La complexité ne réside pas dans le processus de fabrication prédéterminé mais plutôt dans la combinatoire qui naît de la prise en compte des limitations de ressources. Ce chapitre dresse un panorama de la classification et de la résolution des problèmes d'atelier. Nous nous focaliserons en particulier sur le problème d'atelier à cheminements libres et nous ne discuterons que des méthodes évoquées par la suite pour les autres problèmes d'atelier. L'objectif principal est de donner les clés pour la compréhension des chapitres 6 et 7 présentant notre méthode de résolution.
Ce chapitre est organisé de la manière suivante. La section 4.1 définit les problèmes d'atelier et présente leur classification. Ensuite, les sections 4.2 et 4.3 récapitulent respectivement les principaux résultats de complexité et méthodes de résolution. Finalement, la section 4.4 situe nos axes de recherche par rapport à la littérature.

4.1 Définition des problèmes d'atelier

Nous donnons dans cette section une définition des problèmes d'atelier et introduisons une classification avant d'illustrer ces problèmes par quelques applications réelles.

4.1.1 Définition des problèmes de base

Dans un problème d'atelier, une pièce doit être usinée ou assemblée sur différentes machines. Chaque machine est une ressource disjonctive, c'est-à-dire qu'elle ne peut exécuter qu'une tâche à la fois, et les tâches sont liées exclusivement par des contraintes d'enchaînement. Plus précisément, les tâches sont regroupées en n entités appelées travaux ou lots. Chaque lot est constitué de m tâches à exécuter sur m machines distinctes. Il existe trois types de problèmes d'atelier, selon la nature des contraintes liant les tâches d'un même lot. Lorsque l'ordre de passage de chaque lot est fixé et commun à tous les lots, on parle d'atelier à cheminement unique (*flow-shop*). Si cet ordre est fixé mais propre à chaque lot, il s'agit d'un atelier à cheminements multiples (*job-shop*). Enfin, si le séquencement des tâches des travaux n'est pas imposé, on parle d'atelier à cheminements libres (*open-shop*). Un critère d'optimalité souvent étudié est la minimisation du délai total de l'ordonnancement (*makespan*). Ce critère est particulièrement intéressant puisque les ordonnancements au plus tôt constituent des sous-ensembles dominants[1] pour de nombreux critères réguliers. De plus, l'analyse des ordonnancements au plus tôt permet de déterminer des chemins critiques, c'est-à-dire des chemins sur lesquels tout retard a des conséquences sur toute la chaîne, ou des goulots d'étranglement, c'est-à-dire les étapes qui vont limiter la production de tout l'atelier.
Nous supposons que les durées d'exécution des tâches sont données par une matrice entière $P : m \times n$, dans laquelle $p_{ij} \geq 0$ est la durée d'exécution de la tâche $T_{ij} \in T$ du lot J_j réalisée sur la machine M_i.
Une borne inférieure classique pour les problèmes d'atelier, notée C_{max}^{LB}, est égale au maximum de la charge des machines et des durées des lots :

$$C_{max}^{LB} = \max\left(\max_{1\leq i\leq m}\left(\sum_{j=1}^{n} p_{ij}\right), \max_{1\leq j\leq n}\left(\sum_{i=1}^{m} p_{ij}\right)\right).$$

La figure 4.1 illustre un ordonnancement optimal d'un problème d'open-shop où chaque ligne correspond à l'ordonnancement d'une machine alors que les tâches d'un même lot sont identifiées par leur couleur.

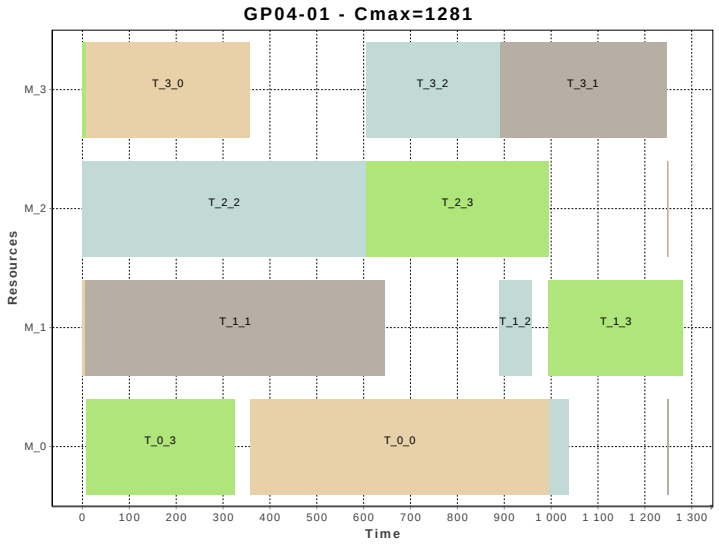


FIGURE 4.1 – Un ordonnancement optimal de délai total 1170 pour l'instance d'open-shop `GP04-01`.

4.1.2 Variantes et classification de Lawler

Les variantes des problèmes d'atelier imposent généralement des restrictions supplémentaires sur l'ordonnancement (contraintes temporelles, contraintes de ressource) ou autorisent la préemption des tâches. Le problème d'open-shop ne contient pas, comme le job-shop ou le flow-shop, de contraintes de précédence

1. Un ensemble de solutions d'un problème d'optimisation est dit dominant s'il contient au moins une solution optimale.

entre les tâches d'un même lot. Cependant, il peut exister des précédences entre les lots.
Dans le cas préemptif, l'exécution d'une opération T_{ij} peut être interrompue par l'exécution d'une autre tâche. Elle sera alors terminée plus tard, éventuellement après d'autres interruptions. Notons que dans l'open-shop, la tâche préemptée peut appartenir au même lot, ce qui n'est pas permis dans les autres problèmes d'atelier à cause des contraintes de précédence.
Enfin, dans certains problèmes appelés problèmes d'atelier sans attente (*no-wait problems*), les lots doivent être exécutés d'une seule traite, sans attente entre les machines. En d'autres termes, il n'y a pas de préemption des lots. Cette contrainte peut être due à l'absence de capacité de stockage intermédiaire ou au processus de fabrication lui-même.

Nous utiliserons la classification des problèmes d'ordonnancement suivant la notation $\alpha|\beta|\gamma$ proposée par Graham *et al.* [87] et étendue par Lawler *et al.* [88]. Notons o le symbole vide. Par souci de clarté, nous négligerons la double indexation des tâches T_{ij} dans un atelier.

- α permet de spécifier l'environnement machine : $\alpha = \alpha_1; \alpha_2$.
  - $\alpha_1 = O$ pour l'open-shop, F pour le flow-shop et J pour le job-shop.
  - α_2 correspond au nombre de machines, o si ce nombre n'est pas fixé.
- β décrit les caractéristiques des tâches : $\beta = \beta_1; \beta_2; \beta_3; \beta_4; \beta_5; \beta_6$.
  - $\beta_1 = pmtn$ dans le cas d'un problème préemptif, o sinon.
  - $\beta_2 = prec$ si des contraintes de précédence existent entre les lots, *tree* si le graphe de précédences est une arborescence, o s'il n'existe pas de contraintes de précédence.
  - $\beta_3 = r_j$ si des dates de disponibilité sont associées aux tâches, o sinon.
  - $\beta_4 = d_j$ si des échéances sont associées aux tâches, o sinon.
  - $\beta_5 = (p_j = p)$ si les tâches ont des durées identiques, $(p_j = 1)$ si les tâches ont des durées unitaires, o sinon.
  - $\beta_6 = no\text{-}wait$ pour les problèmes d'atelier sans attente, o sinon.
- γ est le critère d'optimalité du problème. Notons C_j les dates d'achèvement des tâches.
  - $\gamma = f_{max} (= \max f_j)$ où $f_j(t)$ est la fonction de coût de la tâche j en fonction de sa date d'achèvement t. toutes les fonctions de coûts étudiées sont régulières, c'est-à-dire f_j est une fonction non décroissante de t pour $1 \leq i \leq n$.
  - $\gamma = C_{max}$ $(= \max C_j)$ si l'on cherche à minimiser le délai total ou date d'achèvement maximale (*makespan*).
  - $\gamma = \sum C_j$ si l'on veut minimiser le délai moyen (*flow time*).
  - $\gamma = \sum w_j C_j$ si l'on veut minimiser le délai moyen pondéré (*weighted flow time*).
  - $\gamma = L_{max}$ $(= \max(C_j - d_j))$ si l'on veut minimiser le retard algébrique maximal (*maximum lateness*).
  - $\gamma = \sum U_j$ si l'on veut minimiser le nombre de tâches en retard (*number of late jobs*). La fonction U_j indique si la tâche j est en retard (la valeur de U_j est 1 si la tâche j est en retard $(C_j > d_j)$ et égale à 0 autrement).

Une fonction de coût f_j est dite additive si $f_j(t + \Delta) = f_j(t) + f_j(\Delta)$ pour tout Δ et incrémentale si $f_j(t + \Delta) = f_j(t) + \Delta$. Le délai moyen pondéré d'une tâche est une fonction additive, alors que son retard algébrique est une fonction incrémentale.

4.1.3 Applications

Dans la réalité, un lot correspond généralement à un produit qui doit subir des opérations sur différentes machines. Un exemple classique est le problème d'ordonnancement d'un garage dans lequel des voitures (lots) doivent passer sur différents postes de travail (machines) comme la vidange, le graissage, le lavage ... On retrouve aussi des problèmes de ce type dans l'accomplissement de formalités administratives (passage à différents guichets), dans l'organisation d'examens médicaux de patients hospitalisés, dans la maintenance d'avion lors d'une escale, en télécommunications ... Ces applications peuvent comporter des contraintes de séquencement ou pas : un examen radiologique doit être programmé avant une consultation avec un spécialiste ; un étudiant doit récupérer une convention de stage et un relevé de notes dans un ordre quelconque. Beaucoup de problèmes d'emploi du temps peuvent être vus comme des problèmes d'open-shop avec des contraintes de partage de ressource supplémentaires.

4.2 Résultats de complexité

De nombreux travaux ont été effectués sur la complexité des problèmes d'atelier. Très peu de problèmes d'atelier peuvent être résolus en temps polynomial. Dans la quasi-totalité des cas polynomiaux, les algorithmes sont conçus pour construire des ordonnancements de durée C_{max}^{LB}. Les cas les plus connus sont les problèmes à deux machines ou préemptifs.

Nous allons résumer les principaux résultats connus à ce jour. Nous nous focaliserons sur le délai total C_{max}. Nous donnerons cependant quelques résultats concernant d'autres critères.

4.2.1 Problèmes à deux machines

Parmi les problèmes d'open-shop non préemptifs, le problème $O2||L_{max}$ dans lequel on cherche à minimiser le retard algébrique maximal des tâches est un problème NP-difficile *au sens fort* [89], de même que le problème $O2||\sum C_j$ [90]. Par contre, le problème $O2||C_{max}$ admet un algorithme linéaire O(n) proposé par Gonzalez et Sahni [91] qui construit un ordonnancement optimal sans temps mort de durée C_{max}^{LB}. Il construit séparément deux ordonnancements pour une partition des lots correspondant à la relation $p_{1i} \geq p_{2i}$ avant de concaténer ces deux solutions partielles pour former un ordonnancement optimal. Remarquons que cet algorithme est aussi valide dans le cas préemptif. Lawler *et al.* [89] proposent un algorithme en O(n) pour résoudre $O2|pmtn; r_j|C_{max}$.

Un ordonnancement optimal pour le problème $F2||C_{max}$ peut être recherché parmi les ordonnancements de permutation (la séquence d'exécution des lots est identique sur toutes les machines). On applique la condition suffisante d'optimalité donnée par la *règle de Johnson* [92] : le lot i précède le lot j dans la séquence optimale si $\min(p_{i1}, p_{j2}) \leq \min(p_{i2}, p_{j1})$. L'algorithme de Johnson construit un tel ordonnancement en temps polynomial. L'algorithme de Jackson [93] basé sur la règle de Johnson résout le problème $J2||C_{max}$ en temps polynomial. En revanche, $J2|pmtn|C_{max}$ est NP-difficile, même si $F2|pmtn|C_{max}$ peut être résolu en temps polynomial [94]

4.2.2 Problèmes à trois machines : la frontière

Peu de problèmes non préemptifs avec un nombre de machines supérieur à 2 sont polynomiaux. Gonzalez et Sahni [91] démontrent que le problème $O3||C_{max}$ est NP-difficile *au sens faible*. Cependant, certains cas particuliers sont polynomiaux, Adiri et Aizikowitz [95] exploitent l'existence d'une relation de dominance pour construire un ordonnancement optimal à deux machines avant de placer les tâches sur la machine dominée sans créer de conflits.

Un ordonnancement optimal du problème $F3||C_{max}$ peut encore être recherché parmi les ordonnancements de permutation. La règle de Johnson peut être étendue lorsque la seconde machine n'est pas un goulet d'étranglement, c'est-à-dire qu'elle est complètement dominée par une des deux autres machines. Le problème est alors résolu après transformation en un problème à deux machines. Par contre, le problème préemptif $F3|pmtn|C_{max}$ est NP-difficile [94].

4.2.3 Cas général

Graham *et al.* [87] ont démontré la NP-difficulté *au sens fort* de $O||C_{max}$. Le problème devient polynomial lorsque la plus longue tâche est assez courte par rapport à la charge maximale des machines [96]. de Werra [97], de Werra et Solot [98] construisent des ordonnancements optimaux de durée C_{max}^{LB} quand la structure du graphe biparti pondéré associé à l'instance permet le calcul d'une coloration en temps polynomial. Les problèmes $F||C_{max}$ et $J||C_{max}$ sont aussi NP-difficiles *au sens fort*.

Les problèmes préemptifs sont, en général, plus faciles à résoudre que les non préemptifs car on peut faire appel aux mathématiques du continu. Cho et Sahni [99] utilisent la programmation linéaire pour résoudre le problème $O|pmtn|L_{max}$. Lawler *et al.* [89] proposent un algorithme en O(n) pour résoudre $O2|pmtn; r_j|C_{max}$. Gonzalez et Sahni [91] proposent un algorithme basé sur la construction de couplages de cardinal maximal dans un graphe biparti pour $O|pmtn|C_{max}$. En revanche, nous avons vu précédemment que les problèmes préemptifs de job-shop et flow-shop sont tous deux NP-difficiles pour $m \geq 3$.

Le problème $O|r_j; d_j|C_{max}$ est NP-difficile dans le cas général, même si les tâches ont des durées unitaires sauf dans certains cas particuliers. Graham *et al.* [87] ont démontré que $O|tree|C_{max}$ est NP-difficile au

sens fort pour $m \geq 2$, sauf dans certains cas particuliers où les tâches ont des durées unitaires. Pour tous les critères usuels, les problèmes d'open-shop sans attente sont NP-difficiles *au sens fort*, même les problèmes ne comportant que deux machines [100]. Ces problèmes restent NP-difficiles au sens fort lorsque toutes les tâches ont des durées nulles ou identiques [101].
Il est évident que les problèmes d'atelier avec ressources supplémentaires (c'est-à-dire autres que les machines) ne sont pas plus faciles que les mêmes problèmes sans ressource. Ainsi, tous les résultats de NP-difficulté vus précédemment restent valables pour les problèmes avec ressources. Pire, l'introduction de ressources complique le cas préemptif. de Werra *et al.* [102] ont démontré que le problème $O|pmtn|C_{max}$ avec une ressource renouvelable ou consommable est NP-difficile *au sens fort*.

4.2.4 Conclusion

En résumé, les problèmes d'atelier non préemptif sont NP-difficiles dans le cas général. Le problème d'open-shop préemptif est polynomial contrairement aux problèmes de flow-shop et job-shop, mais il devient NP-difficile lorsque l'on ajoute une ressource. Le lecteur intéressé par la complexité des problèmes d'atelier peut se référer au site complexity results for scheduling problems qui actualise régulièrement ces résultats de complexité.

4.3 État de l'art

Nous allons maintenant présenter les méthodes de résolution pour les problèmes d'atelier non préemptif où l'on cherche à minimiser le délai total, et plus particulièrement pour le problème d'open-shop, le problème de base retenu lors des évaluations des chapitres 6 et 7. L'open-shop a été étudié tardivement par les chercheurs au vu de l'intérêt porté au flow-shop et au job-shop. Les premiers travaux ont principalement concerné les résultats de complexité, l'étude de cas particuliers et la démonstration des garanties de performance pour les heuristiques simples. Depuis, plusieurs approches exactes ou approchées ont été proposées sans toutefois égaler le nombre de travaux portant sur le problème de job-shop. Pour le job-shop, nous nous restreindrons aux dernières approches de la littérature discutées dans le chapitre 7. Par contre, nous n'aborderons pas les approches pour le flow-shop, car ce problème n'a pas été retenu pour nos évaluations.

4.3.1 Jeux d'instances

Trois jeux d'instances pour $O//C_{max}$ sont disponibles dans la littérature. Le premier est constitué de 60 problèmes proposés par Taillard [103] comprenant entre 16 tâches (4 lots et 4 machines) et 400 tâches (20 lots et 20 machines). Il est considéré comme facile puisque la preuve d'optimalité est triviale, c'est-à-dire que la date d'achèvement maximale C_{max} est égale à la borne inférieure C^{LB}_{max}. Brucker *et al.* [104] ont proposé 52 instances difficiles allant de 3 lots et 3 machines à 8 lots et 8 machines. Finalement, le dernier jeu d'instances est constitué de 80 instances proposées par Guéret et Prins [105]. Leurs tailles varient de 3 lots et 3 machines jusqu'à 10 lots et 10 machines et le temps d'achèvement maximal est toujours strictement supérieur à la borne inférieure. Notons que la borne inférieure C^{LB}_{max} est toujours égale à 1000 pour les instances de Brucker et Guéret-Prins.
Taillard [103] a aussi proposé un jeu de 476 instances pour $J||C_{max}$ comprenant 15 ou 20 machines et entre 15 et 100 lots, ainsi qu'un jeu d' instances pour $F||C_{max}$ comprenant 5, 10 ou 20 machines et entre 20 et 500 jobs.

4.3.2 Méthodes approchées

Nous présentons ici les principales heuristiques et métaheuristiques pour les problèmes d'atelier.

4.3.2.1 Heuristiques

Williamson *et al.* [106] ont montré qu'il n'existe pas d'heuristique polynomiale pour $O||C_{max}$ ayant une garantie de performance $\frac{ub}{opt}$ inférieure à $\frac{5}{4}$. On ne dispose pas encore d'heuristique garantie à $\frac{5}{4}$ mais

on peut atteindre une garantie de $\frac{3}{2}$ pour les problèmes à 3 machines.
Une heuristique de liste ou permutation construit un ordonnancement sans temps mort en ajoutant itérativement des tâches à un ordonnancement partiel [107]. Un ordonnancement est dit sans temps mort si aucune machine n'est inactive lorsqu'il est possible d'exécuter un lot. En partant d'un ordonnancement vide, l'algorithme applique, à chaque itération, les étapes suivantes : (a) calculer la date minimale t_0 à laquelle des opérations sont disponibles (il existe au moins un lot et une machine disponibles au temps t_0), (b) ordonnancer une des tâches disponibles à l'aide d'une règle de priorité (*priority dispatching rule*). Dans le cas du flow-shop et du job-shop, une opération est disponible si l'opération la précédant immédiatement dans son lot est achevée. Nous rappelons maintenant quelques-unes des règles de priorité classiques :

random les opérations sont traitées dans un ordre aléatoire,
first fit (FF) les opérations sont traitées dans l'ordre lexicographique,
shortest processing time (SPT) les opérations sont triées par durées croissantes,
longest processing time (LPT) les opérations sont triées par durées décroissantes,
most work remaining (MWR) la priorité est donnée au travail dont la durée résiduelle (durée totale des opérations non traitées) est maximale.

Guéret et Prins [108] et Bräsel *et al.* [109] proposent deux heuristiques originales pour le problème général $O||C_{max}$. La première est basée sur la construction de couplages dans un graphe biparti. La seconde construit un ordonnancement en combinant des techniques d'insertion avec une recherche arborescente tronquée. Dewess *et al.* [110] proposent une méthode de branch-and-bound tronqué travaillant sur l'affectation et la propagation des dates de début au plus tôt et les dates de fin au plus tard.

L'heuristique de la machine goulet, initialement conçue pour $J||C_{max}$, est liée au concept de goulet d'étranglement glissant et à la minimisation du retard algébrique maximum sur une machine par une procédure arborescente pour déterminer un arbitrage complet du graphe disjonctif (voir section 3.4). Pour chaque machine, le problème $1|r_j|L_{max}$ est défini par les fenêtres de temps des tâches après la propagation des précédences du graphe disjonctif pour un ordonnancement au plus tôt. À chaque itération, un branch-and-bound détermine la séquence des opérations de la machine la plus en retard, le goulet d'étranglement, qui est ensuite ajoutée au graphe disjonctif jusqu'à ce qu'il n'y ait plus de conflits (aucune tâche n'est en retard). Des variantes considèrent d'autres critères d'optimalité et types d'atelier, notamment l'open-shop.
Campbell *et al.* [111] proposent une extension multi-étapes de la règle de Johnson générant au plus $m-1$ solutions pour le problème $F||C_{max}$.

4.3.2.2 Métaheuristiques

De nombreuses métaheuristiques pour résoudre l'open-shop ont été développées durant les dernières décennies. Taillard [103] a utilisé une recherche taboue pour résoudre des problèmes carrés de taille 4 à 20 générés aléatoirement. Il propose dans cet article un ensemble de problèmes pour lesquels la recherche taboue fournit des solutions très éloignées de la borne inférieure classique C_{max}^{LB}. Lors des évaluations, nous discuterons de quelques-unes des métaheuristiques les plus récentes et efficaces : un algorithme génétique [112], un algorithme de recherche locale par construction et réparation [113], un algorithme de colonies de fourmis [114] et un algorithme d'essaim de particules [115].
Prins [112] présente quelques algorithmes génétiques dédiés à $O||C_{max}$ basés sur deux idées originales : une population dans laquelle les individus ont des temps d'achèvement maximal distincts ; une procédure spéciale qui réarrange chaque nouveau chromosome.
Chatzikokolakis *et al.* [113] proposent un opérateur générique de réparation basé sur des techniques de recherche locale couplées à une fonction de coût évaluant les affectations partielles.
Une composante majeure de l'optimisation par colonies de fourmis est le mécanisme probabiliste de construction d'une solution. En raison de sa nature constructive, il peut être vu comme une recherche arborescente. De ce fait, Blum [114] combine le mécanisme de construction d'une solution avec un branch-and-bound tronqué en largeur *beam search*. *Beam search* est une méthode incomplète où les affectations partielles sont étendues un nombre limité de fois (cette limite est appelée *beam width*). Cette approche améliora les résultats expérimentaux obtenus par rapport aux meilleurs algorithmes d'optimisation par colonies de fourmis.
L'optimisation par essaim de particules est un algorithme d'optimisation basé sur une population. L'es-

saim est composé de particules qui représentent des solutions distinctes. Sha et Hsu [115] adaptent à l'open-shop la représentation de la position, du mouvement et de la vitesse des particules. Ils atteignent beaucoup de solutions optimales pour les instances de la littérature.

Watson et Beck [116] ont proposé une recherche taboue multi-phases pour le job-shop appelée *i-STS* qui est basée sur l'algorithme de référence *i-TSAB* proposé par Nowicki et Smutnicki [117]. Le but de *i-STS* est de simplifier l'étude de l'influence des diverses composantes de *i-TSAB* sur les performances globales. Cependant, les performances de *i-STS* sont légèrement inférieures à celles de *i-TSAB* sur le jeu d'instance de Taillard au regard de la qualité des solutions pour un même nombre d'itérations.

4.3.3 Méthodes Exactes

Brucker *et al.* [118] ont proposé une recherche arborescente reposant sur la résolution d'un problème à une machine avec des temps d'attente imposés entre les dates de démarrage des tâches. Le principe consiste à transformer le problème d'open-shop en un problème à une machine en mettant bout à bout les m machines. Des temps d'attente minimaux et maximaux sont ensuite imposés entre les dates de début des tâches afin d'être sûr qu'elles seront bien exécutées sur leur machine. Le problème à une machine ainsi obtenu est ensuite résolu par une recherche arborescente dans laquelle les auteurs adaptent les concepts d'arbitrages triviaux et les bornes inférieures utilisés dans la recherche arborescente présentée ci-après.
Brucker *et al.* [104] proposent une recherche arborescente qui arbitre à chaque nœud des disjonctions sur le chemin critique d'une solution heuristique. Malgré l'efficacité de cette méthode, des instances de Taillard ne sont pas résolues à partir de la taille 7×7.
Guéret *et al.* [119] proposent un algorithme avec retour arrière intelligent (*intelligent backtracking*) appliqué à la recherche arborescente de Brucker. Quand une contradiction est levée pendant la recherche, au lieu de revenir systématiquement à la décision précédente (*chronological backtracking*), l'algorithme analyse les causes de la contradiction pour éviter de remettre en question les décisions qui ne sont pas reliées à l'échec afin de revenir sur une décision plus pertinente. Cette approche réduit significativement le nombre de backtracks mais peut s'avérer coûteuse en temps de calcul. Ils appliquèrent plus tard des techniques supplémentaires de réduction de l'arbre de recherche basées sur les intervalles interdits, c'est-à-dire des intervalles temporels dans lesquelles aucune tâche ne peut débuter ou s'achever dans une solution optimale [120].
Dorndorf *et al.* [121] améliorent la recherche arborescente de Brucker par l'usage de techniques avancées de consistance. Au lieu d'analyser ou d'améliorer la stratégie de recherche, ils se concentrent sur l'introduction de techniques de propagation de contraintes pour réduire l'espace de recherche. Ils étudient aussi les performances des procédures d'optimisation top-down et bottom-up présentées au chapitre 2 en fonction de la charge de travail moyenne (*average workload*) de l'instance. Leurs algorithmes furent les premiers à résoudre optimalement un nombre conséquent d'instances en un temps raisonnable. Par contre, certaines instances de Taillard ne sont pas résolues à partir de la taille 15×15 ainsi que des instances de Brucker à partir de la taille 7×7.
Laborie [122] a proposé une recherche arborescente embarquée dans une procédure bottom-up pour la résolution de problèmes d'ordonnancement cumulatif. L'algorithme utilise des techniques avancées d'ordonnancement sous contraintes combinées à des techniques prospectives (*self-adapted shaving*). La stratégie de recherche est basée sur la détection de *minimal critical sets* (MCS), c'est-à-dire des ensembles minimaux de tâches allouées à une même ressource dont l'exécution simultanée entraînerait un dépassement de la capacité. Une heuristique sélectionne le MCS qui maximise la réduction de l'espace de recherche induite par son arbitrage. Cette approche a fermé 34 instances ouvertes de Guéret-Prins et 3 des 6 instances ouvertes de Brucker.
Plus récemment, Tamura *et al.* [31] applique à l'open-shop une méthode d'encodage d'un modèle en satisfaction/optimisation de contraintes composé de clauses sur des contraintes linéaires entières vers un problème de satisfiabilité booléenne (*boolean satisfiability testing problem* – SAT). Les comparaisons $x \leq a$ sont représentées par des variables booléennes différentes pour chaque variable $x \in \mathcal{X}$ et chaque valeur $a \in \mathcal{D}(x)$. Un modèle simple basé sur les contraintes de disjonction entre les tâches partageant un même lot ou une même machine et les contraintes d'achèvement est alors transformé vers SAT. Ils prouvent l'optimalité de toutes les instances dont les trois dernières instances ouvertes de Brucker.

Pour le job-shop au chapitre 7, nous nous appuierons surtout sur les résultats obtenus par Beck [123]

lorsqu'il a proposé la méthode *solution-guided multi-point constructive search* (SGMPCS). SGMPCS est une technique constructive qui réalise une série de recherches arborescentes tronquées en partant d'une affectation vide ou d'une solution découverte précédemment pendant la recherche. Un nombre restreint de solutions d'« élite » est conservé. Après une configuration et une étude poussée de leur algorithme, il montre expérimentalement que SGMPCS domine les principales méthodes constructives, mais reste légèrement moins efficace que la recherche taboue *i-STS*. Récemment, Watson et Beck [116] ont proposé une approche hybride simple entre *i-STS* et SGMPCS qui s'avère compétitive face aux meilleures méthodes de la littérature (dont *i-TSAB*).

4.4 Orientation de nos travaux

Une particularité de l'open-shop, par rapport au flow-shop ou au job-shop, est l'absence de contraintes de précédence. Cette absence augmente la combinatoire du problème et empêche l'adaptation de résultats et algorithmes conçus pour d'autres problèmes d'atelier. Par exemple, on doit considérer un plus grand nombre de disjonctions pour obtenir un arbitrage complet du graphe disjonctif. Par ailleurs, le problème obtenu par transposition des lots et des machines est équivalent au problème original.

La lecture de l'état de l'art montre qu'il n'existe pas d'approche de type top-down basée sur l'ordonnancement sous contraintes qui rivalise avec les meilleures méthodes approchées ou exactes. Or, il est crucial d'obtenir rapidement des solutions comparables à celles des métaheuristiques lors de la résolution de grandes instances, par exemple dans un contexte industriel où l'on assiste à la banalisation des ateliers flexibles. D'autre part, la recherche arborescente de Brucker a été la cible d'un intérêt constant malgré certains défauts notamment sa complexité.

Nous nous concentrerons donc sur le problème d'open-shop de base $O||C_{max}$, même si notre méthode reste valide pour de nombreux autres problèmes de minimisation du temps d'achèvement maximal sans contraintes de partage de ressource supplémentaires ($J||C_{max}$, $F||C_{max}$, $\{O, J, F\}|r_j; d_j; prec|C_{max}$). Notre principal axe de recherche concerne la conception et l'évaluation d'un algorithme pour calculer un arbitrage complet du graphe disjonctif en suivant la procédure top-down qui intègre des techniques de diversification et d'apprentissage. Dans un second temps, nous étudierons les heuristiques de sélection pour l'arbitrage du graphe disjonctif. Nous proposerons de nouvelles heuristiques basées sur les degrés des variables et discuterons de l'influence du modèle sur le comportement des heuristiques d'arbitrage. Finalement, nous décrirons dans le chapitre 9 des techniques de reformulation et de déduction pour la construction automatique du graphe disjonctif à partir d'un modèle quelconque défini par l'utilisateur.

Chapitre 5

Ordonnancement d'une machine à traitement par fournées

Nous rappelons la définition et la classification des problèmes de fournées ainsi que les principaux résultats de complexité. Nous nous intéressons particulièrement aux problèmes de fournées pour lesquels la capacité de la machine est bornée et les dates de disponibilité des tâches sont identiques. Nous établissons ensuite un état de l'art sur ces problèmes.

Sommaire

5.1 Définition des problèmes de fournées . . . 43
5.1.1 Définition du problème de base . . . 44
5.1.2 Variantes et classification de Lawler . . . 45
5.1.3 Applications . . . 45
5.2 Résultats de complexité . . . 45
5.2.1 Modèle en parallèle . . . 46
5.2.2 Modèle en série . . . 46
5.2.3 Conclusion . . . 46
5.3 État de l'art . . . 47
5.3.1 Jeu d'instances . . . 47
5.3.2 Méthodes de résolution . . . 47
5.4 Orientation de nos travaux . . . 48

L'ordonnancement de fournées consiste à exploiter au mieux des moyens limités, une machine à traitement par fournées, pour organiser un ensemble de tâches. La complexité de ce problème ne réside pas seulement dans l'ordonnancement des fournées, mais aussi dans leur construction. Ce chapitre dresse un panorama de la classification et de la résolution des problèmes de fournées. L'objectif principal est de donner les clés pour la compréhension du chapitre 8 présentant notre approche pour le problème de base. Ce chapitre est organisé de la manière suivante. La section 5.1 définit le problème de base et introduit la classification des problèmes de fournées. Ensuite, les sections 5.2 et 5.3 récapitulent respectivement les principaux résultats de complexité et méthodes de résolution. Finalement, la section 5.4 situe nos axes de recherche par rapport à la littérature.

5.1 Définition des problèmes de fournées

Nous donnons dans cette section une définition des problèmes de fournées et introduisons leur classification avant d'illustrer ces problèmes par quelques applications réelles.

5.1.1 Définition du problème de base

Le problème de base consiste à trouver un ordonnancement de n tâches sur une machine à traitement par fournées minimisant une fonction objectif régulière. Une machine à traitement par fournées peut exécuter simultanément plusieurs tâches. En ordonnancement, une fonction objectif est dite régulière lorsqu'elle est croissante en fonction des dates d'achèvement des tâches. Des tâches exécutées simultanément forment une fournée. Toutes les tâches d'une fournée débutent et s'achèvent aux mêmes instants, car leurs dates de début et de fin sont celles de la fournée à laquelle elles appartiennent. Aucune tâche ne peut être ajoutée ou retirée de la fournée pendant son exécution. Nous supposerons que les tâches et la machine sont disponibles depuis l'instant 0 ou, ce qui est équivalent, que leurs dates de disponibilité sont identiques.
Plus précisément, nous nous intéresserons au modèle *burn-in* ou *paralell-batch* ou encore *p-batch*, dans lequel la durée d'exécution d'une fournée est égale à celle de la plus longue tâche appartenant à cette fournée. Il existe deux variantes du modèle *p-batch* : le modèle non borné (*unbounded model*), dans lequel $b > n$ implique qu'une fournée peut contenir un nombre quelconque de tâches ; le modèle borné (*bounded model*), dans lequel $b < n$ implique qu'une fournée peut contenir un nombre limité de tâches. Le cas particulier $b = 1$ est équivalent à problème d'ordonnancement sur une machine, puisque la machine ne peut exécuter qu'une tâche à chaque instant. De ce fait, le modèle borné donne naissance à des problèmes qui sont au moins aussi difficiles que leurs homologues traditionnels à une machine.
Dans cette thèse, nous considérons une extension du modèle borné dans laquelle les tâches ont des tailles différentes définie formellement de la manière suivante. On considère un ensemble J de n tâches et une machine à traitement par fournées de capacité b. Chaque tâche j est caractérisée par un triplet d'entiers positifs (p_j, d_j, s_j), où p_j est sa durée d'exécution, d_j est sa date échue, et s_j est sa taille. La machine à traitement par fournées peut traiter plusieurs tâches simultanément tant que la somme de leurs tailles ne dépasse pas sa capacité b. Toutes les données sont déterministes et connues a priori. La date de fin C_j d'une tâche est la date de fin de la fournée à laquelle elle appartient. Le critère d'optimalité étudié est la minimisation du retard algébrique maximal : $L_{max} = \max_{1 \leq j \leq n}(C_j - d_j)$. Ce problème est NP-difficile *au sens fort* puisque Brucker *et al.* [124] ont prouvé que le même problème avec des tâches de tailles identiques était NP-difficile *au sens fort*. Le problème étudié dans cette thèse est noté $1|p\text{-}batch; b < n; non\text{-}identical|L_{max}$ en suivant la classification présentée dans la section suivante.
La figure 5.1 illustre un ordonnancement optimal pour un notre problème de fournées où le retard maximal est égal à -90, c'est-à-dire que toutes les tâches sont en avance. La machine à traitement par fournées est représentée par un diagramme dans lequel les axes horizontaux et verticaux correspondent respectivement au temps et à la charge de la ressource. La capacité de la machine à traitement par fournées est $b = 10$.

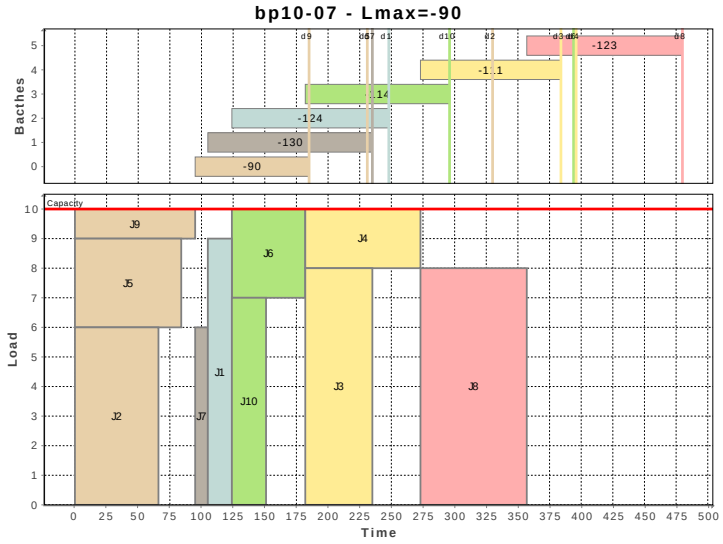


FIGURE 5.1 – Un ordonnancement optimal de retard maximal -90 pour l'instance `bp10-07`.

Une tâche est dessinée comme un rectangle dont la longueur et la hauteur représentent respectivement sa durée et sa taille. Les tâches dont les dates de début sont identiques appartiennent à la même fournée. La solution contient donc six fournées. Le retard algébrique d'une fournée est représenté dans la partie haute de la figure par un rectangle allant de sa date échue à sa date d'achèvement. Dans cet exemple, la première fournée contenant les tâches 2, 5 et 9 détermine la valeur de l'objectif.

5.1.2 Variantes et classification de Lawler

Nous présentons une classification des problèmes de fournées et complétons la notation $\alpha|\beta|\gamma$ présentée en section 4.1 pour ces problèmes.

- α permet de spécifier l'environnement machine.
  - $\alpha = 1$, car nous ne considérons que des problèmes à une seule machine.
- β décrit les caractéristiques des tâches et des diverses autres restrictions : $\beta = \beta_1; \beta_2; \beta_3; \beta_4$. Nous ne discuterons que des problèmes sans contrainte de précédence ($prec$), ni date de disponibilité (r_j).
  - $\beta_1 =$ *p-batch* ou *parallel-batch* lorsque la durée d'exécution d'une fournée est égal à la plus longue durée d'exécution de ses tâches, sinon *s-batch* ou *serial-batch* lorsque la durée d'exécution d'une fournée est égale à la somme des durées d'exécution de ses tâches.
  - $\beta_2 = b < n$ dans le cas du modèle borné (*p-batch*), o dans le cas du modèle non borné.
  - $\beta_3 = (p_j = p)$ si les tâches ont des durées identiques, $(p_j = 1)$ si les tâches ont des durées unitaires, o sinon. Outre les modèles *p-batch* et *s-batch*, Webster et Baker [125] considère un troisième modèle correspondant à *p-batch*$; p_j = p$, dans lequel la durée d'exécution d'une fournée est constante et indépendante de ses tâches.
  - $\beta_4 =$ *non-identical* si les tâches ont des tailles différentes dans le modèle borné ($b < n$), o sinon.
- γ est un des critères d'optimalité présenté en section 4.1. Nous nous limiterons aux critères d'optimalité pour lesquels l'homologue à une machine $1||\gamma$ peut être résolu en temps polynomial : f_{max}, C_{max}, $\sum C_j$, $\sum w_j C_j$, L_{max}, $\sum U_j$.

Dans le modèle en série (*s-batch*), l'exécution de tâches avec des caractéristiques différentes entraîne un temps d'attente $s > 0$ sur la machine à traitement par fournées. Ces temps d'attente représentent par exemple un nettoyage de la machine ou un changement d'outils. Le *family scheduling model* est une variante du modèle en série (*s-batch*) assez répandue où les tâches sont partitionnées en familles pour lesquelles il n'y a aucun temps d'attente entre deux tâches d'une famille. Une fournée correspond alors à une séquence maximale de tâches sans temps d'attente. Le lecteur intéressé pourra consulter l'état de l'art de Potts et Kovalyov [126] pour de plus amples informations sur les problèmes de fournées.

5.1.3 Applications

La recherche sur les problèmes de fournées est motivée par l'industrie (chimie, pharmacie, aéronautique et électronique), où l'utilisation de fours, de séchoirs ou encore d'autoclaves[1] est fréquente. Par exemple, Lee *et al.* [127] utilise le modèle *p-batch* en réponse aux problématiques existantes dans la fabrication à grande échelle de circuits intégrés.

Le problème étudié dans cette thèse est issu de l'industrie aéronautique. Les pièces en matériaux composites utilisées dans cette industrie sont constituées de deux catégories de matériaux : une matrice (résine *epoxy*) et un renfort (fibre de carbone). Elles sont fabriquées selon le processus suivant : la matrice est recouverte de fibre de carbone, puis la pièce résultante est consolidée à haute température et à haute pression dans un autoclave (la machine à traitement par fournées). Différentes pièces de tailles différentes (les tâches) peuvent être traitées simultanément dans un même autoclave.

5.2 Résultats de complexité

Nous rappelons les principaux résultats de complexité pour les problèmes de machine à traitement par fournées sans contrainte de précédence, ni date de disponibilité.

1. Un autoclave est un récipient à fermeture hermétique destiné à la cuisson ou à la stérilisation.

5.2.1 Modèle en parallèle

5.2.1.1 Modèle non borné

Nous supposons ici que la machine à traitement par fournées peut exécuter simultanément un nombre quelconque de tâches ($1|p\text{-}batch|\gamma$). Le problème de minimisation du délai total est trivialement résolu en plaçant toutes les tâches dans la même fournée. Le délai total est alors la valeur maximale des durées d'exécution des tâches. Brucker *et al.* [124] donnent une caractérisation d'une classe dominante d'ordonnancement pour les critères réguliers : supposons que les tâches sont triées par durée non décroissante (*shortest processing time (SPT-)rule*), il existe au moins un ordonnancement optimal qui peut être déterminé en précisant uniquement les tâches qui commencent chaque fournée, car il est possible de reconstruire la séquence complète de fournées en suivant la règle SPT. Un algorithme générique en programmation dynamique se base sur cette caractérisation pour la minimisation d'une fonction de coût de la forme $\sum f_j$. Les complexités temporelles et spatiales de cet algorithme sont pseudo-polynomiales et valent respectivement $O(n^2P)$ et $O(nP)$ où P est la somme des durée d'exécution des tâches. Le problème NP-difficile *au sens fort* $1|p\text{-}batch|\sum f_j$ est moins complexe que son homologue à une machine $1|p\text{-}batch|\sum f_j$ qui est NP-difficile *au sens fort*. Cette caractérisation est à la base d'algorithmes polynomiaux pour certaines fonctions objectif. Les auteurs proposent un algorithme de complexité $O(n^2)$ pour minimiser le délai moyen pondéré $\sum w_jC_j$, un autre de complexité $O(n^3)$ pour la minimisation du nombre de jobs en retard $\sum U_j$ et un dernier de complexité $O(n^2)$ pour minimiser le retard algébrique maximal L_{max}. Ce dernier algorithme est utilisé comme une sous-procédure pour construire un algorithme polynomial minimisant le coût maximal d'une fonction de coût régulière f_{max}.

5.2.1.2 Modèle borné

Concernant le modèle borné ($1|p\text{-}batch; b < n|\gamma$), Brucker *et al.* [124] ont montré qu'il existe un algorithme de complexité $\min\{O(n\log n), O(\frac{n^2}{b}\}$ pour la minimisation du délai total C_{max} et un algorithme de programmation dynamique de complexité pseudo-polynomiale $O(n^{b(b-1)})$ pour la minimisation du délai moyen $\sum C_j$. En outre, Brucker *et al.* montrent que les critères d'optimalité L_{max}, f_{max} et $\sum U_j$ engendrent des problèmes NP-difficiles *au sens fort*. Cependant, lorsque toutes les tâches ont des durées identiques ($p_j = p$), Baptiste [128] montre que les critères d'optimalité L_{max}, $\sum C_j$, $\sum w_jC_j$ et $\sum U_j$ engendrent des problèmes polynomiaux. Pour finir, la complexité engendrée par les critères d'optimalité $\sum C_j$ pour b quelconque et $\sum w_jC_j$ pour b fixé ou quelconque reste une question ouverte.

Pour le modèle borné où les tâches de tailles différentes ($1|p\text{-}batch; b < n; non\text{-}identical|\gamma$), tous les critères d'optimalité engendrent des problèmes NP-difficiles *au sens fort*. Uzsoy [129] a prouvé que les critères C_{max} et $\sum C_j$ engendrent des problèmes NP-difficiles *au sens fort*. Le calcul de la complexité engendrée par les critères d'optimalité L_{max}, f_{max} et $\sum U_j$ est immédiat puisque nous venons de voir que les mêmes problèmes avec des tâches de tailles identiques étaient NP-difficiles *au sens fort*.

5.2.2 Modèle en série

Pour le modèle en série ($1|s\text{-}batch|\gamma$), le problème de minimisation du délai total est trivialement résolu en plaçant toutes les tâches dans la même fournée. Ainsi, le délai total est la somme des durée d'exécution des tâches. Coffman *et al.* [130] ont proposé un algorithme de complexité $O(n\log n)$ pour la minimisation du délai moyen $\sum C_j$. Ng *et al.* [131] ont proposé un algorithme de complexité $O(n^2)$ pour la minimisation du retard algébrique maximal L_{max}. Brucker et Kovalyov [132] ont proposé un algorithme de complexité $O(n^3)$ pour la minimisation du nombre de tâches en retard $\sum U_j$. Par contre, le délai moyen pondéré $\sum w_jC_j$ engendre un problème NP-difficile *au sens fort*. Cependant, Baptiste [128] ont montré que ce problème devient polynomial lorsque les tâches ont des durées identiques ($p_j = p$). À notre connaissance, la complexité engendrée par le critère d'optimalité f_{max} reste une question ouverte.

5.2.3 Conclusion

En conclusion, le tableau 5.1 récapitule les résultats de complexité pour les problèmes de fournées sans contrainte de précédence ni date de disponibilité en se restreignant aux critères d'optimalité pour

lesquels les homologues à une machine sont polynomiaux ($b = 1$ voir Lawler *et al.* [88]). Pour le modèle non borné, tous les critères d'optimalité engendrent des problèmes polynomiaux. Pour le modèle borné, certains critères engendrent des problèmes polynomiaux lorsque les tailles sont identiques, mais tous deviennent NP-difficiles *au sens fort* lorsque les tâches ont des tailles différentes. Pour le modèle en série, le délai moyen pondéré engendre un problème NP-difficile *au sens fort*, alors que les autres critères engendrent des problèmes polynomiaux. Par ailleurs, Brucker *et al.* [124] a montré que la présence de dates échues ou d'échéances engendre des problèmes NP-difficiles *au sens fort* pour le modèle borné même lorsque $b = 2$.

Fonction Objectif	*parallel-batch*				*serial-batch*
	Non borné	Borné			
	$b > n$	$b = 1$	$b > 1$		
			identical	*non-identical*	
f_{max}	polynomial	$O(n^2)$	NP-difficile	NP-difficile	open
C_{max}	$O(n)$	$O(n)$	$\min\{O(n \log n), O(\frac{n^2}{b}\}$	NP-difficile	$O(n)$
L_{max}	$O(n^2)$	$O(n \log n)$	NP-difficile	NP-difficile	$O(n^2)$
$\sum C_j$	$O(n \log n)$	$O(n \log n)$	$O(n^{b(b-1)})$	NP-difficile	$O(n \log n)$
$\sum w_j C_j$	$O(n \log n)$	$O(n \log n)$	open	NP-difficile	NP-difficile
$\sum U_j$	$O(n^3)$	$O(n \log n)$	NP-difficile	NP-difficile	$O(n^3)$

TABLE 5.1 – Complexité des problèmes de fournées sans contrainte de précédence ni date de disponibilité.

5.3 État de l'art

Les problèmes de fournées n'ont été abordés que récemment dans la littérature. Les premiers résultats ont surtout concerné la complexité de ces problèmes et des algorithmes d'approximation, mais un certain nombre d'approches ont été proposées depuis.

5.3.1 Jeu d'instances

À notre connaissance, il n'existe aucun jeu d'instances standard pour les problèmes de fournées. Dans cette thèse, nous utilisons le jeu d'instances pour $1|p\text{-}batch; b < n; non\text{-}identical|L_{max}$ proposé par Daste *et al.* [133] comprenant entre 10 et 100 tâches. Dans ce jeu d'instances, les durées d'exécution sont uniformément distribuées dans l'intervalle entier $[1, 99]$ ce qui correspond à des applications dans la fabrication de semi-conducteurs. Les tailles des tâches sont uniformément distribuées dans l'intervalle entier $[1, 10]$ et la machine à traitement par fournées a une capacité b égale à 10 (inspiré par les travaux de Ghazvini et Dupont [134]). Pour une instance donnée, on détermine d'abord les durées d'exécution et les tailles de toutes les tâches avant de générer les dates échues des tâches en se basant sur une formule inspirée par les travaux de Malve et Uzsoy [135] : $d_j = U[0, \alpha] \times \tilde{C}_{max} + U[1, \beta] \times p_j$ où $\tilde{C}_{max} = (\sum_{j=1}^{n} s_j \times \sum_{j=1}^{n} p_j) \div (b \times n)$ est une approximation (borne inférieure) du délai total nécessaire à l'exécution de toutes les tâches sur la machine à traitement par fournées. Pour un nombre de tâches $n \in \{10, 20, 50, 75, 100\}$, 40 instances ont été générées avec les paramètres $\alpha = 0.1$ et $\beta = 3$.

5.3.2 Méthodes de résolution

La première étude semble être la publication de l'article d'Ikura et Gimple [136] en 1986, qui présente un algorithme exact en $O(n)$ pour le problème $1|p\text{-}batch; b < n; r_j; p_j = p|C_{max}$, c'est à dire un problème où les durées d'exécution et les tailles des tâches sont identiques, les tâches ont des dates de disponibilité, et le critère d'optimalité est le délai total de l'ordonnancement. Depuis, différentes approches ont été proposées pour la minimisation de critères d'optimalité dépendant des dates échues dans un problème où les tâches ont une taille identique : des heuristiques [137–139] ; un algorithme génétique [138] ; des

méthodes exactes [125, 140, 141]. Plusieurs articles traitent aussi de problèmes avec des tâches de tailles différentes, mais les critères d'optimalité étudiés ne dépendent que des dates d'achèvement des tâches (C_{max}, $\sum C_j$, $\sum w_j C_j$) : des heuristiques [129, 142] ; un algorithme génétique [143] ; une méthode par recuit simulé [144] ; des méthodes exactes [129, 134, 142, 145–149].
Malgré le succès de la programmation par contraintes sur des problèmes d'ordonnancement variés, il existe, à notre connaissance, une seule approche par contraintes pour ce type de problème : les algorithmes de filtrage de Vilím [150] pour un problème *s-batch* avec des familles de tâches et des temps d'attente dépendant de la séquence. Ces algorithmes étendent les algorithmes de filtrage de la contrainte de partage de ressource disjonctive (voir section 3.3.1), qui raisonne principalement sur les dates d'achèvement des tâches et le délai total de l'ordonnancement.
À notre connaissance, seuls deux articles [133, 151] traitent de la résolution d'un problème avec des tâches de tailles différentes et un critère d'optimalité dépendant des dates échues. Dans ces travaux, Daste *et al.* proposent une formulation en programmation mathématique basée sur la construction et le séquencement des fournées [133], puis une approche par *branch-and-price* [151]. Une approche par branch-and-price intègre un branch-and-bound et des méthodes de génération de colonne pour résoudre des problèmes en nombres entiers de grande taille [152]. L'approche par branch-and-price est basée sur un problème maître où chaque colonne représente une fournée. Une solution du problème maître est une séquence réalisable de fournées. À chaque itération, on résout un sous-problème pour déterminer une colonne (fournée) qui améliore la solution du problème maître. Ce sous-problème est résolu par un algorithme glouton, puis si nécessaire, par une méthode exacte d'énumération.
Pour une synthèse plus poussée sur les problèmes de fournées, les lecteurs intéressés pourront se reporter à l'article de Potts et Kovalyov [126].

5.4 Orientation de nos travaux

Les problèmes avec des tâches de tailles différentes et un critère d'optimalité dépendant des dates échues n'a jamais été abordé en programmation par contraintes, notre premier axe de recherche concerne donc la modélisation de notre problème. Nous nous efforcerons de proposer un modèle concis et élégant en nous appuyant sur la richesse du langage déclaratif de modélisation et la flexibilité des algorithmes de recherche.
Une particularité du problème de fournées étudié dans cette thèse est que son homologue à une machine, noté $1||L_{max}$, peut être résolu en temps polynomial [153] : un ordonnancement optimal est obtenu en appliquant la règle de Jackson, aussi appelée règle EDD (*earliest due date (EDD-)rule*) qui séquence les tâches en ordre non décroissant de date échue [154]. Nous avons vu dans le chapitre 2 que les contraintes globales sont souvent utilisées pour modéliser un sous-problème de manière concise et efficace. Elles raisonnent généralement sur la faisabilité du sous-problème, mais dans le cadre d'un problème d'optimisation, les réductions de domaine peuvent également être réalisées sur la base des coûts. On supprime pendant la propagation des combinaisons de valeurs qui ne peuvent pas être présentes dans une solution améliorant la meilleure solution (ou borne supérieure) découverte jusqu'à présent. Focacci *et al.* [155] ont proposé d'embarquer ces raisonnements dans une contrainte globale qui représente une relaxation d'un sous-problème ou d'elle-même. Ils ont montré l'intérêt d'utiliser cette information pour le filtrage, mais aussi pour guider la recherche sur plusieurs problèmes d'optimisation combinatoire. Notre second axe de recherche consiste à exploiter l'homologue à une machine de notre problème de fournées pour améliorer la résolution de notre modèle.

Deuxième partie

Contribution

Chapitre 6

Ordonnancement d'atelier au plus tôt

Ce chapitre présente une approche en ordonnancement sous contraintes pour la résolution exacte du problème d'open-shop ($O//C_{max}$) basée sur les toutes dernières techniques de propagation et de recherche combinées à une borne supérieure initiale obtenue de manière heuristique. Les techniques de randomisation et redémarrage combinées à l'enregistrement de nogoods favorisent la diversification de la recherche tout en évitant les redondances d'un redémarrage à l'autre. Cette approche domine les meilleures métaheuristiques et méthodes exactes sur de nombreuses d'instances.

Sommaire

6.1 Modèle en ordonnancement sous contraintes . . . 52
6.1.1 Contraintes disjonctives . . . 52
6.1.2 Contraintes temporelles . . . 52
6.1.3 Contraintes supplémentaires . . . 53
6.1.4 Stratégie de branchement . . . 54
6.2 Algorithme de résolution . . . 55
6.2.1 Principe de l'algorithme . . . 55
6.2.2 Solution initiale . . . 57
6.2.3 Techniques de redémarrage . . . 57
6.3 Évaluations expérimentales . . . 58
6.3.1 Réglage des paramètres . . . 59
6.3.2 Analyse de sensibilité . . . 60
6.3.3 Comparaison avec d'autres approches . . . 64
6.4 Conclusion . . . 67

Ce chapitre présente nos travaux [3, 4] sur les problèmes d'atelier définis au chapitre 4. Dans un problème d'atelier, une pièce doit être usinée ou assemblée sur différentes machines. Chaque machine est une ressource disjonctive, c'est-à-dire qu'elle ne peut exécuter qu'une tâche à la fois. Plus précisément, les tâches sont regroupées en n entités appelées travaux, lots ou jobs. Chaque lot est constitué de m tâches à exécuter sur m machines distinctes. Nous considérerons que le séquencement des tâches appartenant aux lots n'est pas imposé (open-shop). Le critère d'optimalité étudié est la minimisation de le délai total de l'ordonnancement ou date d'achèvement maximale (*makespan*). Notre approche reste valide pour d'autres problèmes d'atelier incluant notamment le job-shop et le flow-shop. Nous supposons que les durées d'exécution des tâches sont données par une matrice entière $P : m \times n$, dans laquelle $p_{ij} \geq 0$ est la durée d'exécution de la tâche $T_{ij} \in T$ du lot J_j réalisée sur la machine M_i.

Nous proposons plusieurs améliorations d'un algorithme top-down dédié aux problèmes d'open-shop qui a l'avantage, contrairement aux approches de type bottom-up, de fournir rapidement de bonnes solutions (voir section 2.4). Notre modèle repose sur des techniques d'inférence forte pour les contraintes disjonctives et temporelles ainsi que des raisonnements dédiés à la minimisation du délai total. Notre

contribution principale est de montrer que la combinaison de techniques de randomisation et redémarrage, et de méthodes d'inférence et de branchement dédiées à l'ordonnancement est une alternative efficace pour la résolution du problème d'open-shop. Les performances de notre algorithme de résolution surpassent celles des approches de la littérature sur un large spectre d'instances.
La section 6.1 introduit notre modèle en ordonnancement sous contraintes et la section 6.2 décrit les techniques améliorant l'algorithme top-down. Finalement, la section 6.3 présente nos résultats expérimentaux et étudie l'influence des différentes composantes de l'algorithme avant de le comparer aux autres approches.

6.1 Modèle en ordonnancement sous contraintes

Dans cette section, nous présentons notre modèle en ordonnancement sous contraintes pour le problème $O//C_{max}$. Nous précisons les règles et algorithmes choisis pour le filtrage de la contrainte disjonctive représentant les machines et les lots. Puis, nous expliquons la gestion du filtrage des contraintes temporelles et du branchement dans le modèle disjonctif. Nous présentons deux contraintes globales supplémentaires pour la gestion des intervalles interdits (*forbidden intervals*) et l'élimination d'une symétrie (*symmetry breaking*). Pour finir, nous détaillerons plusieurs stratégies de branchement en ordonnancement sous contraintes et justifions notre choix de branchement.

6.1.1 Contraintes disjonctives

La contrainte disjonctive présentée en section 3.3.1 représente une ressource de capacité unitaire. La contrainte est satisfaite si aucune paire de tâches de durée non nulle ne se chevauche. Une contrainte disjonctive est postée pour chaque lot et chaque machine.

Nous avons choisi l'implémentation proposée par Vilím [43] des règles *not first/not last*, *detectable precedence* et *edge finding*. Nous tirons avantage du calcul de la date d'achèvement minimale ECT_Ω (*earliest completion time*) des tâches d'un ensemble Ω pour estimer une borne inférieure sur le délai total de l'ordonnancement. En fait, la date d'achèvement minimale d'une machine M_i (respectivement un lot J_j) est égale à la valeur ECT_{M_i} (respectivement ECT_{J_j}). Par conséquent, le délai total de l'ordonnancement est supérieur aux dates d'achèvement minimales des ressources (lots et machines).

6.1.2 Contraintes temporelles

Nous rappelons que le graphe de précédence $\mathcal{G} = (\mathcal{T}, \mathcal{P})$ (voir section 3.4) est associé à un sous-réseau de contraintes où ses sommets représentent les tâches et les arcs représentent les contraintes de précédence. Deux tâches fictives T_{start} et T_{end} représentant le début et la fin de l'ordonnancement sont ajoutées. Un arc entre deux tâches T_{ij} et T_{kl} est ajouté à $\mathcal{P}$ si T_{ij} précède T_{kl} ($T_{ij} \preceq T_{kl}$). Les arcs initiaux de $\mathcal{P}$ sont tous ceux d'origine T_{start} ou de destination T_{end}. Il suffit d'ajouter les arcs correspondant aux séquences d'opérations dans des lots et de supprimer les contraintes disjonctives sur les lots pour modéliser les problèmes de job-shop et flow-shop.

La stratégie de branchement que nous allons utiliser ajoute des arcs entre les paires de tâches partageant une ressource (lot ou machine) jusqu'à ce que toutes ces paires soient connectées par un chemin de $\mathcal{G}$. Il s'agit donc d'un arbitrage partiel du graphe disjonctif qui induit un arbitrage complet par transitivité. À la fin de la recherche, le délai total de l'ordonnancement C_{max} est égal à la longueur du plus long chemin entre T_{start} et T_{end}, c'est-à-dire un chemin critique. La stratégie de branchement exploite la fermeture transitive de $\mathcal{G}$ pour éviter de créer des cycles ou de considérer les précédences transitives ou obligatoires. En effet, le graphe de précédence $\mathcal{G}$ est consistent si et seulement s'il ne contient aucun cycle.Une précédence est facilement détectée en raisonnant sur ces fenêtres de temps (règles précédences interdites et obligatoires), mais cela n'est pas nécessairement le cas lorsque la précédence est impliquée par transitivité. Les précédences respectent l'inégalité triangulaire. Par conséquent, si un arc (T_{ij}, T_{kl}) est transitif, c'est-à-dire que T_{ij} et T_{kl} sont reliés par un chemin dans $\mathcal{P}\backslash\{(T_{ij}, T_{kl})\}$, alors la précédence $T_{ij} \preceq T_{kl}$ est déduite.

Nous utilisons une contrainte globale modélisant le graphe de précédence introduit en section 3.4 pour le filtrage et le branchement. Bien sûr, les contraintes temporelles peuvent être gérées par l'ajout

des contraintes arithmétiques élémentaires associées. Mais, il existe des procédures efficaces dédiées à un problème voisin appelé *simple temporal problem* [36]. Ce problème considère un ensemble de variables temporelles entières $\{X_1, \ldots, X_n\}$ et un ensemble de contraintes $\{a_{ij} \leq X_j - X_i \leq b_{ij}\}$ où $b_{ij} \geq a_{ij} \geq 0$. Cesta et Oddi [156] ont proposé plusieurs algorithmes pour la gestion des informations temporelles permettant de (a) gérer dynamiquement l'ajout et le retrait de contraintes dans le réseau, et (b) exploiter la structure du réseau pour la propagation incrémentale et la détection des cycles.
La contrainte globale gère efficacement l'ajout et le retrait d'arcs dans le graphe de précédence. Son état est restauré lors d'un retour arrière, c'est-à-dire qu'elle maintient une pile représentant des changements du graphe. Ainsi, cette contrainte ne réalise aucune hypothèse sur l'ensemble des disjonctions à arbitrer, c'est-à-dire l'ensemble des arêtes du graphe disjonctif. Lors de l'ajout d'un arc, il est possible de détecter en temps constant sa transitivité ou la création d'un cycle en maintenant la fermeture transitive de $\mathcal{G}$. Frigioni *et al.* [157] ont proposé un algorithme pour maintenir la fermeture transitive d'un graphe orienté avec une complexité $O(n)$ pour une série d'insertions et de suppressions d'arcs. En outre, nous maintenons un ordre topologique de $\mathcal{G}$ grâce à l'algorithme simple et efficace proposé par Pearce et Kelly [158]. Notez que la connaissance de la fermeture transitive diminue la complexité totale pour maintenir l'ordre topologique.
La propagation des précédences est réalisée en temps linéaire par la contrainte globale alors que la propagation individuelle des précédences par le solveur peut atteindre le point fixe en temps quadratique si l'ordre de réveil des contraintes est malheureux. Dans notre cas, les plus longs chemins entre T_{start} et T_{ij}, et entre T_{ij} et T_{end} sont calculés pour mettre à jour la fenêtre de temps de la tâche T_{ij}. Puisque $\mathcal{G}$ est un graphe orienté acyclique, tous les plus longs chemins issus de T_{start} et terminant à T_{end} sont calculés en temps linéaire par une version incrémentale de l'algorithme *dynamic bellman algorithm for the single source longest path problem* [37]. Notre implémentation maintient dynamiquement un ordre topologique qui est une entrée de l'algorithme pour éviter des calculs redondants. À chaque appel, notre contrainte globale ne considère qu'un sous-graphe induit par les tâches dont les fenêtres de temps ont changé depuis le dernier appel. Ainsi, la complexité totale est réduite à $O(|\mathcal{P}|)$ par rapport à l'algorithme général en $O(|\mathcal{T}| \times |\mathcal{P}|)$ proposé par Cesta et Oddi [156].

6.1.3 Contraintes supplémentaires

Nous introduisons des contraintes supplémentaires qui améliorent la réduction des domaines en considérant le critère d'optimalité et une symétrie basique. Ces contraintes redondantes basées sur des règles de dominance ne sont pas nécessaires à la correction du modèle mais améliorent sa résolution. Elles peuvent être propagées en temps constant durant la recherche contrairement à d'autres bornes inférieures complexes ou règles de dominance appliquées par d'autres stratégies de recherche [159], comme la stratégie de branchement de Brucker.

6.1.3.1 Intervalles interdits

Les intervalles interdits sont des techniques de filtrage exclusivement dédiées à la minimisation du délai total de l'ordonnancement dans un atelier à cheminement libre. Les intervalles interdits sont des intervalles temporels dans lesquels aucune tâche ne peut débuter ou s'achever dans une solution optimale. Les fenêtres de temps des tâches sont ajustées pendant la recherche à partir de ces informations. Quand la date de début au plus tôt d'une tâche appartient à un intervalle interdit, elle peut être augmentée jusqu'à la borne supérieure de cet intervalle. Guéret et Prins [120] déterminent les intervalles interdits grâce à la résolution de $m + n$ problèmes *subset sum* (voir section 3.5). Les problèmes sont résolus avec une complexité $O(d \times n)$ [voir 63] où d représente, dans notre cas, le délai total de l'ordonnancement. Les fenêtres de temps sont ajustées en temps constant puisque ces problèmes sont résolus une fois pour toutes avant la résolution.

6.1.3.2 Élimination d'une symétrie

De nombreux problèmes de satisfaction de contraintes contiennent des symétries qui définissent des classes d'équivalence pour de nombreuses solutions. Les techniques d'élimination des symétries réduisent la découverte de solutions équivalentes en évitant de visiter les affectations partielles symétriques de

celles prouvées inconsistantes ou dominées. Une solution du problème $O//C_{max}$ peut être inversée en considérant la dernière tâche comme la première, la seconde comme l'avant-dernière et ainsi de suite. Ainsi, une solution obtenue par réflexion a la même délai total que la solution originale. Par conséquent, il n'est pas nécessaire de vérifier l'ordre inverse lorsque l'algorithme a prouvé qu'un ordre était sous-optimal. On peut briser cette symétrie par réflexion en choisissant une tâche quelconque T_{ij} et en imposant qu'elle débute dans la première moitié de l'ordonnancement : $s_{ij} \leq \left\lceil \frac{e_{\text{end}} - p_{ij}}{2} \right\rceil$. Notre algorithme sélectionne la première tâche dont la durée d'execution est maximale en suivant l'ordre lexicographique.

6.1.4 Stratégie de branchement

Il existe deux grandes catégories de stratégies de branchement en ordonnancement sous contraintes : l'affectation des dates de début aux tâches et l'arbitrage des disjonctions. La première catégorie mène à un branchement n-aire alors que la seconde engendre généralement des décisions binaires.

6.1.4.1 Affectation des dates de début

Parmi les branchements de la première catégorie, la stratégie classique *dom-minVal* (voir section 2.3) est généralement vouée à l'échec dans le cas d'optimisation d'un critère régulier en ordonnancement sous contraintes. En effet, l'ensemble des ordonnancements actifs (aucune tâche ne peut terminer plus tôt sans en retarder au moins une autre) est dominant, c'est-à-dire qu'il contient au moins une solution optimale. Une stratégie bien connue appelée *setTimes* [160] est incomplète dans le cas général, mais complète pour les critères réguliers. À chaque nœud, elle choisit une tâche parmi l'ensemble des tâches non ordonnancées et disponibles puis l'ordonnance au plus tôt. Lors d'un retour arrière, la stratégie marque la tâche ordonnancée au dernier point de choix comme indisponible jusqu'à ce que sa date de début au plus tôt change (par propagation).

6.1.4.2 Arbitrage des disjonctions

La seconde catégorie arbitre les disjonctions, c'est-à-dire qu'elle impose des précédences entre certaines paires de tâches. Le branchement n-aire proposé par Brucker *et al.* [104], noté *block*, est basé sur le calcul d'une solution heuristique (compatible avec l'arbitrage partiel courant) à chaque nœud pour déterminer quelles précédences ajouter. La stratégie exploite un théorème de dominance pour remettre en cause les précédences du chemin critique de cette solution heuristique en imposant dans chaque branche plusieurs précédences entre les tâches de ce chemin critique. Cette stratégie fixe simultanément de nombreuses précédences tout en restant complète.

Beck *et al.* [161] ont proposé un branchement binaire noté *profile* dans lequel une disjonction entre deux tâches critiques partageant la même ressource disjonctive est arbitrée à chaque nœud. Pour ce faire, il définit la demande individuelle comme la quantité (probabiliste) de ressource nécessaire à une activité à chaque instant t. Pour mesurer l'occupation d'une ressource, on additionne les courbes des demandes individuelles de ses tâches. À chaque nœud, on détermine le pic de demande sur l'ensemble des ressources. On identifie alors la paire de tâches qui participe le plus à la demande de la ressource à l'instant du pic de demande (ressource – instant) et l'on s'assure que la disjonction associée n'est pas arbitrée. L'ordre relatif des tâches doit alors être déterminé pour la disjonction choisie. Cette étape est réalisée par l'heuristique randomisée de sélection d'arbitrage appelée *centroid* [161]. Le centroid est une fonction déterministe du domaine vers les réels qui est calculée pour les deux tâches critiques. Le centroid est l'instant qui divise la demande individuelle en deux parties d'aires égales. On impose d'abord la précédence qui préserve l'ordre des centroids des deux tâches. Si les centroids sont placés au même instant, un ordre aléatoire est choisi. Plus récemment, Laborie [122] a proposé un branchement générique pour l'ordonnancement cumulatif basé sur la notion d'ensemble critique minimal (*minimal critical set* – MCS). En ordonnancement disjonctif, les MCSs sont simplement des paires d'activités partageant une ressource disjonctive. À chaque nœud, la stratégie consiste à (a) choisir un MCS en fonction d'une estimation de la réduction de l'espace de recherche engendré par sa résolution (b) appliquer une procédure de simplification sur le MCS choisi et (c) brancher sur les différentes précédences possibles jusqu'à ce qu'il ne reste plus aucun MCS.

Les exemples donnés en figure 6.1 illustrent les différentes formes de l'arbre de recherche en fonction de la stratégie de branchement. Dans ce chapitre, nous utiliserons les heuristiques de sélection *profile* et

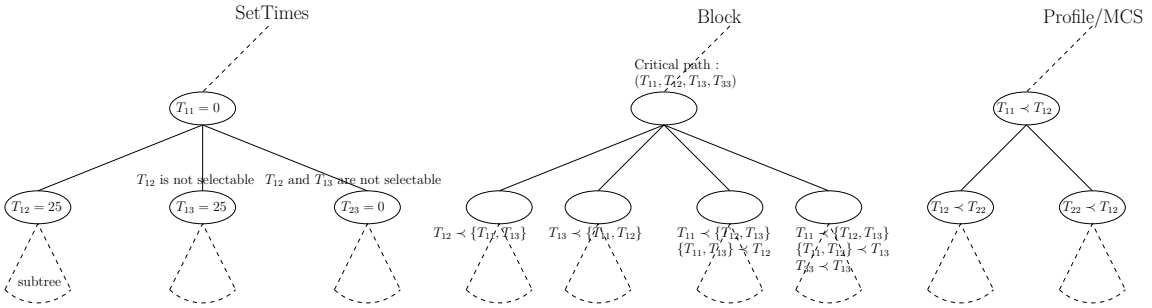


FIGURE 6.1 – La forme de l'arbre de recherche pour différentes stratégies de branchement. De gauche à droite, les stratégies *setTimes*, *block* et *profile*/MCS.

centroid dont la randomisation des points de choix binaires facilite l'application des techniques introduites en section 6.2.

6.2 Algorithme de résolution

En utilisant le modèle présenté en section 6.1, on peut maintenant résoudre le problème d'open-shop par la procédure d'optimisation top-down (voir section 2.4). Cette section décrit les améliorations apportées à l'algorithme de recherche par l'utilisation d'heuristiques et de techniques de redémarrage. La stratégie de branchement consiste à ajouter des contraintes de précédences en chaque nœud grâce à l'heuristique de sélection de disjonction *profile* et à l'heuristique randomisée de sélection d'arbitrage *centroid*. Des expérimentations préliminaires ont révélé deux principaux inconvénients à cette approche. Premièrement, les techniques de filtrage sont très efficaces dès qu'une borne supérieure précise est connue, mais ne font que ralentir la recherche dans le cas contraire. Ensuite, la légère randomisation de *centroid* provoque des variations importantes du temps de résolution, mais aussi de la qualité des premières solutions découvertes. Un tel comportement laisse supposer que certaines des premières décisions prises dans l'arbre de recherche ne sont jamais remises en question et provoque un phénomène de *thrashing* important (le même échec est découvert plusieurs fois).

Nous proposons une heuristique constructive randomisée (sans propagation) pour initialiser la borne supérieure initiale afin d'améliorer la sélection et la propagation des premières décisions de l'arbre de recherche. De plus, nous appliquons une stratégie de redémarrage dans l'espoir de diversifier la recherche et nous enregistrons des *nogoods* pour éviter d'explorer la même partie de l'espace de recherche d'un redémarrage à l'autre. Ces mécanismes peuvent être intégrés dans n'importe quel algorithme top-down et n'importe quelle stratégie de branchement même si l'enregistrement des *nogoods* est simplifié dans le cas d'un branchement binaire. Nous donnons maintenant une présentation détaillée de notre approche et discutons ses paramètres.

6.2.1 Principe de l'algorithme

La figure 6.2 récapitule notre algorithme générique pour la résolution des problèmes d'atelier : *Randomized and Restart Constraint Programming algorithm* (`RRCP`). L'algorithme de type top-down débute par le calcul d'une solution heuristique (blocs 1 et 2) et continue, si nécessaire, par une recherche complète (blocs 3 à 9) qui améliore itérativement la meilleure solution découverte jusqu'à présent.
L'heuristique calcule une solution initiale qui fournit la borne supérieure initiale C_{max}^{UB}. Nous avons élaboré une heuristique randomisée simple appelée `CROSH` présentée en section 6.2.2. Puis, un test d'optimalité (bloc 2) détermine s'il est nécessaire de démarrer la recherche complète. Dans le bloc 3, le modèle PPC défini en section 6.1 et divers composants du solveur sont initialisés.

La boucle principale de l'algorithme est exécutée entre les blocs 4 et 9. Après la réduction des domaines par propagation (bloc 4), l'algorithme a atteint une solution, une contradiction, ou ni l'une ni l'autre. Si une solution est découverte, elle est enregistrée et la nouvelle borne supérieure génère une coupe dynamique (bloc 5) qui est propagée pendant les retours en arrière (bloc 6). Si une contradiction est

levée et que toutes les branches du nœud racine ont été visitées, alors l'optimalité de la dernière solution découverte est prouvée entraînant l'arrêt de l'algorithme. Dans le cas contraire, l'algorithme effectue un retour arrière (bloc 6). À ce moment, la création d'un nouveau point de choix est nécessaire, mais nous examinons d'abord la possibilité de redémarrer. Nous avons retenu deux politiques de redémarrage (Luby et Walsh en bloc 7) qui sont présentées en section 6.2.3. Avant un redémarrage, nous enregistrons des *nogoods* (bloc 8) issus du chemin vers la dernière branche de l'arbre pour éviter une exploration redondante d'un redémarrage à l'autre et garder ainsi la trace des sous-problèmes prouvés sous-optimaux ou inconsistants. Un *nogood* est défini ici par la borne supérieure courante et l'ensemble des décisions de branchement (précédences). L'algorithme ne redémarre jamais avant un retour arrière pour éviter des effets de bord tels que l'enregistrement de *nogoods* incomplets ou un redémarrage après l'obtention de la preuve d'optimalité (par exemple à la racine de l'arbre). Lorsque l'algorithme ne redémarre pas, la recherche arborescente continue par la création d'un nouveau point de choix (bloc 9) en suivant la stratégie de branchement *profile*. La stratégie de branchement divise le problème principal en deux sous-problèmes disjoints en ajoutant temporairement une contrainte de précédence (branche gauche) puis son opposé (branche droite).
Notre algorithme contient donc deux paramètres : le nombre d'itération de l'heuristique construisant une solution initiale (bloc 1) ; la stratégie de redémarrage (bloc 7). Leurs valeurs sont discutées en section 6.3.1.

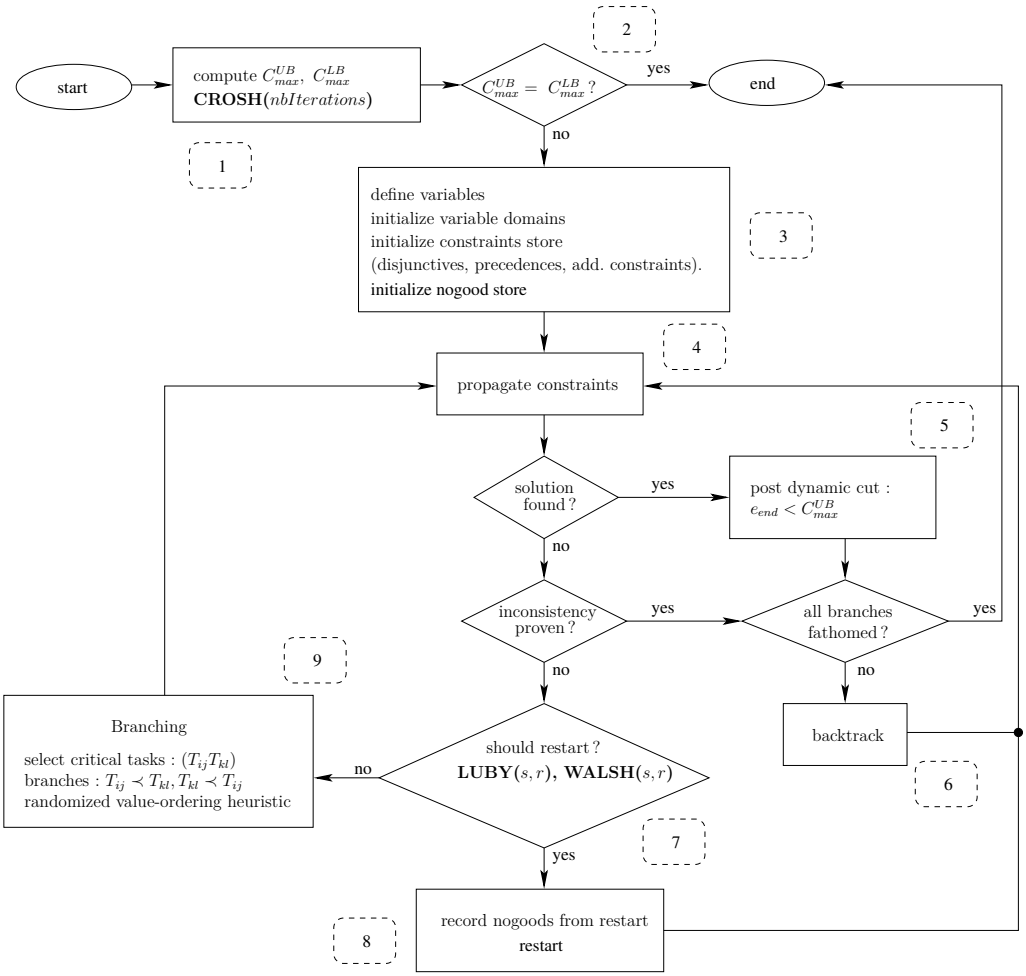


FIGURE 6.2 – Structure de RRCP. Les ellipses sont les états finaux et initiaux. Les rectangles sont des procédures ou actions. Les losanges sont des conditions *if-else*. Les rectangles en pointillé sont des étiquettes.

6.2.2 Solution initiale

Le filtrage avec inférence forte est coûteux, mais devient utile seulement lorsque les fenêtres de temps des tâches ne sont pas trop relâchées. En l'absence de contraintes de disponibilité et d'échéance, les fenêtres de temps dépendent fortement de la borne supérieure sur le délai total. De la même manière, la stratégie de branchement est très sensible aux fenêtres de temps puisque la demande individuelle et le *centroid* des tâches en dépendent. Par conséquent, il est important d'obtenir rapidement une borne supérieure de qualité sur le délai total de l'ordonnancement.

Nous avons élaboré une heuristique constructive randomisée pour le problème d'open-shop (*Constructive Randomized Open-Shop Heuristics* – `CROSH`) en combinant des règles de priorité pour les heuristiques de liste (voir section 4.3). En plus d'être une heuristique générique et simple à implémenter, les performances de `CROSH` se sont avérées tout à fait satisfaisantes (voir section 6.3.2). La première itération ordonnance les opérations en suivant la règle de priorité LPT dans laquelle les opérations sont triées par durées décroissantes. Les itérations suivantes traitent les opérations dans un ordre aléatoire sous une distribution uniforme. Le seul paramètre est le nombre d'itérations. La complexité d'une itération est égale à $O(m^2 \times n^2)$.

6.2.3 Techniques de redémarrage

Les politiques de redémarrage sont basées sur l'observation suivante : plus une recherche arborescente avec retour arrière passe de temps sans trouver de solution, plus il est probable qu'elle explore une partie stérile de l'espace de recherche. Les choix initiaux effectués par le branchement sont les moins informés et les plus importants puisqu'ils mènent aux plus grands sous-arbres et que l'exploration peut rarement remettre en cause les erreurs précoces. Cela peut entraîner des situations de *thrashing* où des échecs dus à un ensemble restreint de choix initiaux sont redécouverts plus profondément dans les sous-arbres.
Pour contrer cet inconvénient, les techniques de *shaving* et d'*intelligent backtracking* (voir section 2.2) ont été étudiées pour la résolution de problèmes d'atelier [119, 121, 122]. Le *shaving* essaie d'affecter une valeur à une variable et applique un algorithme de consistance. Si une inconsistance est détectée, la valeur peut être supprimée du domaine de la variable. Les algorithmes avec retour arrière intelligent essaient de compenser les erreurs initiales du branchement en analysant les échecs et identifiant les décisions responsables de l'échec courant.
Nous avons choisi d'incorporer dans notre approche des stratégies de redémarrage combinées à une heuristique d'arbitrage randomisée pour réduire le *thrashing* et remettre en cause les mauvais choix initiaux. Cette approche diversifie la recherche, nécessite moins de calculs en chaque nœud, mais explore plus de nœuds que le *shaving* et l'*intelligent backtracking*. Des expérimentations préliminaires sur le *shaving* ont montré qu'il n'était pas utile dans notre implémentation. Nous n'avons pas testé d'algorithme avec retour arrière intelligent parce qu'il requiert un mécanisme d'explication de tous les changements des domaines ainsi qu'une intégration profonde dans l'algorithme de recherche.
Nous rappelons d'abord la définition d'une politique universelle de redémarrage. Ensuite, nous décrivons le mécanisme d'enregistrement des *nogoods* qui évitent l'exploration de la même partie de l'espace de recherche d'un redémarrage à l'autre.

6.2.3.1 Politique universelle de redémarrage

Soit $A(x)$ un algorithme randomisé de type *Las Vegas*, c'est-à-dire que pour n'importe quelle entrée x, le résultat de A est toujours correct, mais son temps d'exécution $T_A(x)$ est une variable aléatoire. Une stratégie universelle de redémarrage (*universal restart strategy*) détermine la limite des longueurs de toutes les exécutions de l'algorithme pour n'importe quelle distribution des temps d'exécution.

Si la seule observation réalisable est la longueur d'une exécution et si l'on ne dispose d'aucune information sur la distribution des temps d'exécution $T_A(x)$, Luby *et al.* [162] ont montré qu'une planification universelle des longueurs limites de la forme $(1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, \ldots)$ donne un temps espéré de résolution qui est à un facteur logarithmique de celui donné par la meilleure limite fixe. De plus, le facteur d'amélioration des performances dû à une autre stratégie universelle est au mieux constant. Nous considérons deux paramètres qui sont le facteur d'échelle s (*scale factor*) et le facteur géométrique r (*geometric factor*). Le facteur d'échelle est la limite de base d'une stratégie de redémarrage. Notons

$\lambda_k = \frac{r^k-1}{r-1}$, le i-ème terme de la séquence est ($s = 1$ et $r = 2$ dans l'exemple précédent) :

$$\forall i > 0, \quad t_i = \begin{cases} sr^{k-1} & \text{si } i = \lambda_k \\ t_{i-\lambda_{k-1}} & \text{si } \lambda_{k-1} < i < \lambda_k \end{cases}$$
$$s = 2 \text{ and } r = 3 \Rightarrow \quad 2, 2, 2, 6, 2, 2, 2, 6, 2, 2, 2, 6, 18, \ldots$$

Walsh [163] a suggéré une autre stratégie universelle de la forme $s, sr, sr^2, sr^3, \ldots$ où l'accroissement des limites est exponentiel contrairement à l'accroissement linéaire de la stratégie de Luby.

Wu et van Beek [164] ont démontré analytiquement et empiriquement les lacunes liées à l'emploi de stratégies qui ne sont pas universelles. Ils ont aussi montré que le paramétrage des stratégies améliore les performances tout en conservant les garanties d'optimalité et de pire cas. Les redémarrages sont une composante clé de notre approche, nous évaluerons donc l'influence des facteurs d'échelle et géométrique pour identifier une bonne stratégie de redémarrage.

6.2.3.2 Enregistrement des nogoods avant les redémarrages

La sélection d'une disjonction est déterministe, mais peut varier d'un redémarrage à l'autre en fonction de l'évolution de la borne supérieure et des décisions d'arbitrage. En effet, la sélection de l'arbitrage est randomisée, mais seulement lorsqu'on ne peut pas identifier un ordre jugé prometteur grâce à *centroid*. Nous avons déjà évoqué les variations importantes de la résolution provoquées par cette légère randomisation. Quelquefois, au contraire, peu d'arbitrages aléatoires sont rendus et la même partie de l'arbre de recherche est souvent visitée d'un redémarrage à l'autre. Nous appliquons donc une technique d'enregistrement des *nogoods* avant les redémarrages similaire à celle de Lecoutre *et al.* [165] pour pallier cet inconvénient.

Dans notre cas, un *nogood* est défini pour la borne supérieure courante *ub* et correspond à un ensemble de précédences P, tel que toute solution associée à un arbitrage complet contenant P a un délai total supérieur à *ub*. Le même ensemble P de précédences peut être redécouvert d'un redémarrage à l'autre. L'enregistrement de P permet d'éviter une exploration redondante et de provoquer une meilleure diversification au gré des redémarrages. L'enregistrement des *nogoods* est peu intrusif puisqu'il a lieu juste avant les redémarrages (bloc 8 de la figure 6.2). À ce moment, on enregistre tous les *nogoods* représentant les sous-arbres prouvés sous-optimaux en suivant l'idée de Lecoutre *et al.*. Par conséquent, l'exploration accomplie durant cette étape est totalement mémorisée et cette partie de l'arbre de recherche ne sera plus visitée lors des étapes suivantes. Un nombre linéaire de *nogoods*, au regard du nombre de disjonctions, est enregistré à chaque redémarrage puisqu'un *nogood* est extrait de chaque décision négative (branche droite) du chemin allant vers la dernière branche de l'arbre de recherche binaire.

Les *nogoods* sont propagés individuellement dans Lecoutre *et al.* grâce aux techniques de *watch literals*. Nous avons implémenté une contrainte globale qui assure la propagation unitaire des *nogoods*. La propagation unitaire consiste à simplifer les clauses lorsqu'un littéral est instancié : chaque clause le contenant est supprimée ; la négation du littéral est supprimée de chaque clause la contenant. Notre implémentation reste naïve et pourrait être améliorée par les techniques de *watch literals* [166]. En pratique, le nombre de *nogoods* est relativement bas puisqu'ils ne sont enregistrés qu'avant les redémarrages et leur propagation ne revêt pas un aspect critique dans notre approche. De plus, nous supprimons les *nogoods* qui sont subsumés par les nouveaux venus lors de leur enregistrement avant un redémarrage.

6.3 Évaluations expérimentales

Trois jeux d'instances pour $O//C_{max}$ sont disponibles dans la littérature. Le premier est constitué de 60 problèmes proposés par Taillard [103] comprenant entre 16 tâches (4 lots et 4 machines) et 400 tâches (20 lots et 20 machines). Il est considéré comme facile puisque la preuve d'optimalité est triviale, c'est-à-dire que la date d'achèvement maximale C_{max} est égale à la borne inférieure C^{LB}_{max}. Brucker *et al.* [104] ont proposé 52 instances difficiles allant de 3 lots et 3 machines à 8 lots et 8 machines. Finalement, le dernier jeu d'instances est constitué de 80 instances proposées par Guéret et Prins [105]. Leurs tailles varient de 3 lots et 3 machines jusqu'à 10 lots et 10 machines et le temps d'achèvement maximal est

toujours strictement supérieur à la borne inférieure. Notons que la borne inférieure C_{max}^{LB} est toujours égale à 1000 pour les instances de Brucker et Guéret-Prins.

Nous avons réalisé différentes expérimentations dans le but de (a) configurer les paramètres (section 6.3.1) (b) étudier l'impact des différentes composantes de l'algorithme (section 6.3.2) et (c) comparer RRCP aux meilleures approches de la littérature (section 6.3.3). Nous avons élaboré deux jeux d'expérimentations indépendants pour réaliser l'étape (a) en un temps raisonnable. En utilisant les meilleurs paramètres identifiés à l'étape (a), les résultats du principal jeu d'expérimentations servent de support aux étapes (b) et (c). Dans ce jeu principal, plusieurs variantes de l'algorithme sont appliquées sur toutes les instances. Ces variantes déterminent la solution initiale avec CROSH ou LPT et utilisent une recherche arborescente avec, ou sans, stratégie de redémarrage (Luby/Walsh) qui enregistre, ou pas, des *nogoods*. L'algorithme étant randomisé, chaque instance est résolue 20 fois sans limite de temps. Les temps de résolution prennent en compte ceux de l'heuristique fournissant la solution initiale. Notre implémentation est basée sur le solveur Choco (Java), notamment son module d'ordonnancement (tâches, contraintes de partage de ressource et temporelles, branchements), et celui pour les redémarrages avec enregistrement des *nogoods*. Étant donné que ces fonctionnalités ont été intégrées dans les livraisons récentes de choco ($\geq$ 2.0.0), notre algorithme peut être facilement reproduit (voir chapitre 9). Un module additionnel fournit les heuristiques, construit le modèle et configure le solveur.
Toutes les expérimentations ont été menées sur une grille de calcul composée de machines Linux, dans laquelle chaque nœud possède 1 GB de mémoire vive et un processeur AMD cadencé à 2.2 GHz.

6.3.1 Réglage des paramètres

Les paramètres de notre algorithme RRCP sont présentés en section 6.2. Une étude expérimentale justifie ici le choix des paramètres dans le réglage final.

6.3.1.1 Solution initiale

Dans cette section, nous discutons comment fixer le nombre d'itérations de CROSH signalé en bloc 1 de la figure 6.2. Ce jeu d'expérimentations tente de trouver un bon compromis entre le temps passé à calculer la solution heuristique et la qualité de celle-ci. Idéalement, on souhaite arrêter l'heuristique dès que la recherche complète peut améliorer cette solution plus rapidement ou prouver son optimalité.

Par conséquent, nous avons discrétisé le nombre d'itérations selon l'ordre de grandeur 1, 10, 100, 1000, 5000, 10000, 25000. Le nombre maximum d'itérations est limité à 25000 parce que les temps d'exécution de CROSH dépassent une limite globale de 30 secondes pour les plus grandes instances ($15 \times 15,\ 20 \times 20$). Toutes les instances sont résolues avec une variante de l'algorithme sans redémarrage et dont la solution initiale est déterminée par CROSH. Chaque instance est résolue 20 fois avec une limite de temps de 180 secondes.

Nous avons déduit le nombre d'itérations pour chaque taille de problème à partir du pourcentage d'instances résolues optimalement et du temps moyen de résolution d'une instance. Le nombre d'itérations des instances dont la taille est inférieure à 6×6 est fixé à 1000 puisque celles-ci sont résolues facilement par la recherche complète. Ensuite, le nombre d'itérations est fixé à 10000 jusqu'à la taille 9×9 et à 25000 autrement. En complément, une limite de temps globale (indépendante de la taille des instances) est fixée à 20 secondes.

6.3.1.2 Paramètres des politiques de redémarrage

Cette section traite du réglage des paramètres des stratégies de redémarrage signalés au bloc 7 de la figure 6.2 (facteurs d'échelle et géométrique). Nous avons sélectionné un ensemble de 23 instances dont la taille est comprise entre 6×6 et 20×20 (8 GP*, 9 j*, 6 tai*) et dont les distributions des temps d'exécution sont différentes pour identifier les bonnes valeurs des paramètres. Nous mesurons l'influence des paramètres sur l'efficacité des stratégies de redémarrage en considérant le nombre de problèmes résolus optimalement dans une certaine limite de temps comme proposé par Wu et van Beek [164]. Le facteur d'échelle s est discrétisé en ordre de grandeur $10^{-2}, 10^{-1} \ldots,\ 10^{2}$ et le facteur géométrique r en ordre de grandeur $2,\ 3, \ldots,\ 10$ pour Luby et $1.1,\ 1.2, \ldots,\ 2$ pour Walsh. Puis, le facteur d'échelle est multiplié par le nombre de tâches $n \times m$ pour prendre en compte la taille d'une instance. En effet, le facteur d'échelle

est souvent dépendant de la taille, profondeur et largeur, de l'arbre de recherche. Cette multiplication équilibre le nombre de redémarrages pour les différentes tailles des instances. Pour chaque instance, les 20 exécutions des variantes de l'algorithme où la solution initiale est donnée par LPT (toutes les exécutions débutent avec la même solution initiale) ont une limite de temps de 180 secondes.

Le meilleur réglage des paramètres est estimé en choisissant celui qui maximise le nombre d'instances résolues optimalement et en cas d'égalité celui qui minimise le temps moyen de résolution d'une instance. Remarquez que les moyennes intègrent les résultats des exécutions pour lesquelles le modèle n'a pas prouvé l'optimalité dans la limite de temps alloué (180 secondes). Ces moyennes sont donc en fait des bornes inférieures. Le tableau 6.1 fournit les résultats des expériences pour les deux stratégies de redémarrage avec enregistrement des *nogoods*. Nous donnons le pourcentage d'instances résolues, le temps moyen de

	Luby				Walsh			
	(s, r).	%	$\bar{t}$	$\bar{n}$	(s, r)	%	$\bar{t}$	$\bar{n}$
Best	(1,3)	82.6	43.1	10055	(1,1.1)	82.6	43.0	9863
Acceptable	(1, *)	82.0	44.5	10347	(0.1,*)	80.2	48.0	10973
Average	(*,*)	75.5	58.6	16738	(*,*)	76.2	54.6	11359
NotAcceptable	(0.01, *)	72.9	67.4	31676	(100, *)	70.5	67.1	12600

TABLE 6.1 – Identification de bons paramètres pour les stratégies de redémarrage avec enregistrement des *nogoods*.

résolution et le nombre moyen de nœuds visités pendant l'exploration pour différents jeux de paramètres. Le symbole * représente l'ensemble des valeurs testées pour un paramètre. La ligne intitulée Best donne les résultats pour les deux paires de paramètres choisis dans le réglage final. La ligne intitulée Average donne les résultats moyens sur l'ensemble des combinaisons de paramètres possibles. Les lignes intitulées Acceptable et NotAcceptable donnent respectivement les résultats moyens de paramétrages où la valeur du facteur d'échelle entraîne une amélioration ou une dégradation significative des performances. Comme espéré, l'estimation de bons paramètres (Best, Acceptable) entraîne une amélioration perceptible des performances par rapport à des stratégies non paramétrées (Average, NotAcceptable). De nombreux paramétrages obtiennent des résultats comparables et leurs performances dépendent principalement de la valeur du facteur d'échelle (par exemple, Acceptable and NotAcceptable). L'équivalence des stratégies de Luby et Wash avec enregistrement des *nogoods* est montrée expérimentalement avec le réglage final de leurs paramètres en section 6.3.2.2. Par conséquent, les autres sections discutent seulement des résultats obtenus avec la stratégie de Luby.

6.3.2 Analyse de sensibilité

Nous rapportons ici les résultats de l'évaluation des techniques présentées en section 6.2 et justifions expérimentalement leur utilisation. Nous rapportons uniquement les résultats des instances dont la taille est supérieure à 6×6 parce que les temps de résolution très courts des petites instances ne sont pas discriminants. Nous rappelons que l'algorithme est exécuté 20 fois pour chaque instance à cause de sa randomisation.

6.3.2.1 Solution initiale

Dans cette section, nous étudions l'influence de `CROSH` sur la résolution des différents jeux d'instances, et comparons `CROSH` à LPT, c'est-à-dire sa première itération. Une analyse préliminaire, non détaillée ici, a montré que `CROSH` exécutait 10000 itérations en moins de 1 seconde pour les instances dont la taille est inférieure à 9×9 et 25000 itérations en moins de 5 secondes pour les instances de taille 10×10. Pour les grandes instances de Taillard, les instances « faciles » sont souvent résolues optimalement en quelques secondes, mais les temps d'exécution montent quelquefois jusqu'à 20 secondes pour atteindre 25000 itérations.

Notons C_{max}^{UB} la borne supérieure sur le délai total associée à la solution initiale fournie par `CROSH`. L'écart à l'optimum est le quotient de la différence entre la borne supérieure et l'optimum sur l'optimum : $\frac{C_{max}^{UB} - C_{max}}{C_{max}}$. Le graphe à gauche de la figure 6.3 illustre la relation entre l'écart à l'optimum de la solution

initiale de CROSH et le temps de résolution de RRCP (donné sur l'axe horizontal). Chaque instance est représentée par un point dont la coordonnée x est le temps moyen de résolution de RRCP (échelle logarithmique) alors que sa coordonnée y est l'écart moyen à l'optimum de CROSH. On constate que la qualité de la solution initiale est satisfaisante puisque l'écart à l'optimum ne dépasse jamais 4% et que les solutions initiales sont optimales ou proches de l'optimum pour de nombreuses instances. On s'aperçoit aussi que le temps de résolution de RRCP n'est pas clairement relié à l'écart à l'optimum de CROSH, notamment pour les instances de Taillard. Par contre, les écarts à l'optimum augmentent pour les instances difficiles de Guéret-Prins et Brucker, alors qu'ils diminuent sur les grandes instances de Taillard. Une analyse plus approfondie, non détaillée ici, a montré que CROSH a découvert l'optimum au moins une fois pour 28 des 40 instances de Taillard, alors que cela n'est arrivé respectivement que pour 6 des 40 instances de Guéret-Prins, et pour 3 des 26 instances de Brucker. De plus, toutes les exécutions ont découvert l'optimum pour 10 grandes instances de Taillard (15×15, 20×20).

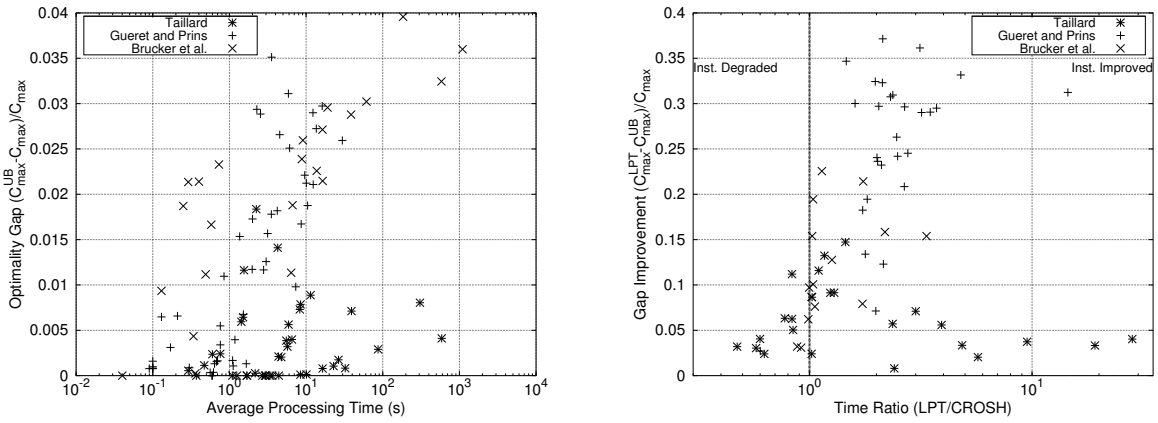


FIGURE 6.3 – Écart à l'optimum de la solution initiale et influence de CROSH sur les performances de RRCP.

Notons $C^{LPT}_{max} \geq C^{UB}_{max}$ la borne supérieure obtenue par LPT (la première itération de CROSH). L'amélioration de l'écart à l'optimum de CROSH par rapport à LPT est estimée en calculant le quotient de la différence entre les bornes supérieures fournies par LPT et CROSH sur l'optimum : $\frac{C^{LPT}_{max} - C^{UB}_{max}}{C_{max}}$. L'amélioration moyenne de l'écart à l'optimum est seulement de 5.7% pour les instances de Taillard, car LPT fournit déjà une borne supérieure précise sur ces instances. Mais, cette amélioration augmente respectivement jusqu'à 10.6% et 25.8% pour les instances de Brucker et Guéret-Prins parce que LPT est moins bien adaptée à ces jeux d'instances.
Le graphe de droite dans la figure 6.3 montre l'influence de CROSH sur notre algorithme. Chaque point représente une instance résolue par des variantes de notre algorithme avec redémarrage et enregistrement des *nogoods*. La coordonnée x est le quotient (échelle logarithmique) du temps moyen de résolution quand la solution initiale est donnée par LPT sur celui obtenu avec une solution initiale fournie par CROSH alors que la coordonnée y est l'amélioration moyenne de l'écart à l'optimum. Les 67 instances dont la taille est strictement supérieure à 6×6 et dont le temps moyen de résolution est compris entre 2 secondes et 1800 pour au moins une variante sont considérées pour dessiner ce graphe. Tous les points situés à droite de la droite ($x = 1 (= 10^0)$) correspondent aux instances dont la résolution est améliorée par l'utilisation de CROSH. L'amélioration des temps de résolution semble liée à celle de l'écart à l'optimum à l'exception des instances de Taillard. En dépit d'améliorations semblables des écarts à l'optimum, certaines instances de Taillard sont résolues 10 fois plus rapidement en utilisant CROSH alors que la résolution des quelques instances situées à gauche de la droite ($x = 1$) est dégradée.

6.3.2.2 Stratégies de redémarrage

Dans cette section, nous étudions l'influence des stratégies de redémarrage de Luby et Walsh sur la résolution et nous montrons expérimentalement leur équivalence dans le contexte du problème d'open-

shop. En utilisant les réglages finaux indiqués dans le tableau 6.1, nous montrons d'abord l'intérêt des redémarrages puis celui d'enregistrer des *nogoods* avant ces redémarrages grâce aux deux graphes de la figure 6.4. Les 61 instances dont la taille est strictement supérieure à 6×6 et dont le temps moyen de résolution est compris entre 2 secondes et 1800 pour au moins une des variantes sont considérées pour dessiner ces graphes. La solution initiale est fournie par `CROSH`.
Le graphe de gauche analyse l'effet sur la résolution des stratégies de redémarrage. Chaque instance est représentée par un point dont la coordonnée x est le quotient du temps moyen de résolution sans redémarrage sur le temps moyen de résolution avec redémarrage et la coordonnée y est le quotient du nombre de nœuds visités sans redémarrage sur le nombre de nœuds visités avec redémarrage. Les échelles des deux axes sont logarithmiques. Tous les points situés au-dessus ou à droite du point (1,1) représentent des instances dont la résolution est améliorée par les redémarrages (quadrant en haut à droite). Au contraire, tous les points situés en dessous ou à gauche du point (1,1) sont des instances dont la résolution est dégradée par les redémarrages (quadrant en bas à gauche). Comme espéré, tous les points se situent autour de la diagonale ($x = y$) puisque le nombre de nœuds visités est à peu près proportionnel au temps de résolution (les quadrants en haut à gauche et en bas à droite sont vides). Les redémarrages améliorent globalement la résolution et certaines instances sont même résolues approximativement 100 fois plus vite grâce aux redémarrages. Par contre, les performances sur un certain nombre d'instances situées en bas à gauche du point (1, 1) sont dégradées.

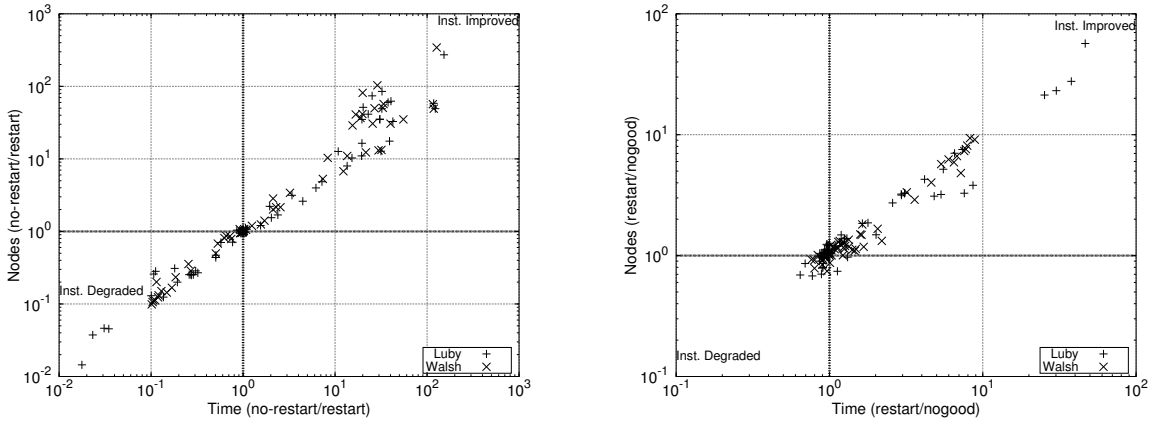


FIGURE 6.4 – Influence des redémarrages (à gauche) et de l'enregistrement des *nogoods* sur les stratégies de redémarrage (à droite).

De la même manière, le graphe de droite montre le gain obtenu par l'enregistrement des *nogoods* avant les redémarrages (les coordonnées de chaque point sont les quotients des temps de résolution et des nombres de nœuds sans et avec enregistrement des *nogoods*). On s'aperçoit que l'enregistrement des *nogoods* améliore la résolution avec redémarrages par un facteur compris entre 1 et 10 pour une grande majorité des instances.
Finalement, en combinant les redémarrages et l'enregistrement des *nogoods*, on obtient les résultats tracés dans le graphe à gauche de la figure 6.5 (les coordonnées de chaque point sont les quotients du temps de résolution et du nombre de nœuds sans redémarrage sur ceux avec redémarrage et enregistrement des *nogoods*). Ainsi, toutes les instances dont la résolution était dégradée par les redémarrages seuls dans la figure 6.4 ont disparu grâce à l'enregistrement des *nogoods*, tout en gardant les effets positifs dus aux redémarrages.
Nous avons montré ici que les redémarrages seuls peuvent améliorer grandement la résolution des problèmes d'open-shop mais manquent de robustesse. En simplifiant, les redémarrages aident à trouver de bonnes solutions rapidement mais, une fois celles-ci connues, la preuve d'optimalité peut demander beaucoup de temps. L'équilibre entre des redémarrages fréquents pour améliorer rapidement la borne supérieure et une exploration plus longue de l'espace de recherche pour prouver l'optimalité est difficile à trouver. L'enregistrement des *nogoods* avant les redémarrages compense cet inconvénient et améliore significativement la résolution comme l'illustre le graphe à gauche de la figure 6.5.

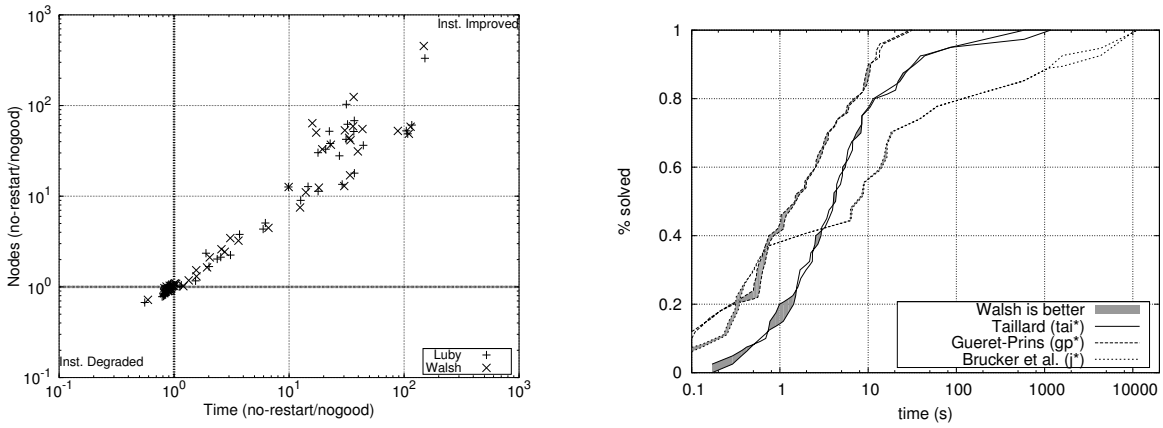


FIGURE 6.5 – Influence des redémarrages avec enregistrement des *nogoods* sur la résolution (à gauche) et équivalence des stratégies de redémarrage avec enregistrement des *nogoods* (à droite).

Finalement, le graphe à droite dans la figure 6.5 montre que les réglages finaux des stratégies de Luby et Walsh obtiennent des performances équivalentes. Pour chaque jeu d'instances, et pour chaque stratégie, le pourcentage d'instances résolues est dessiné comme une fonction du temps de résolution. Les deux courbes associées au même jeu d'instances sont représentées par des lignes dont le motif est identique. L'aire entre ces deux courbes est remplie en gris lorsque la stratégie de Walsh est meilleure que celle de Luby et reste blanche dans le cas contraire. Dans notre cas, les deux stratégies sont équivalentes puisque les deux courbes associées à chaque jeu d'instances sont presque confondues. De plus, ce graphique révèle un ordre croissant de difficulté des jeux d'instances pour notre algorithme : Guéret-Prins $\prec$ Taillard $\prec$ Brucker.

Le tableau 6.2 récapitule les temps moyens de résolution (heure :minute) et le nombre de nœuds visités (millions de nœuds) pour les trois instances les plus difficiles (les temps de résolution sont supérieurs à 1800 secondes). `RRCP` utilise la solution initiale fournie par `CROSH`. On remarque d'abord que l'algorithme sans redémarrage obtient les meilleures performances puisqu'il est nécessaire d'explorer de très nombreux nœuds pour obtenir la preuve d'optimalité. L'enregistrement des *nogoods* revêt alors un aspect critique puisqu'il permet de diviser par trois les temps de résolution. Sans enregistrement des *nogoods*, tous les redémarrages entre la découverte de l'optimum et le dernier sont inutiles puisque l'information est totalement perdue d'un redémarrage à l'autre. Malgré l'enregistrement des *nogoods*, le nombre important de redémarrages, notamment avec la stratégie de Luby, tend à accroître le temps de résolution à cause de la répétition de la sélection et de la propagation des points de choix initiaux. La stratégie de Walsh avec enregistrement des *nogoods* produit des résultats légèrement meilleurs que ceux de la stratégie de Luby. Par contre, son efficacité dépend lourdement de l'enregistrement des *nogoods* puisque les exécutions sont plus longues.

Instance		No Restart		Restart				Nogood Recording			
				Luby		Walsh		Luby		Walsh	
Name	Opt	$\bar{t}$	$\bar{n}$	$\bar{t}$	$\bar{n}$	$\bar{t}$	$\bar{n}$	$\bar{t}$	$\bar{n}$	$\bar{t}$	$\bar{n}$
`j7-per0-0`	1048	1 :43	1.21	5 :56	4.66	18 :10	12.76	2 :10	1.57	2 :03	1.25
`j8-per0-1`	1039	2 :13	1.16	10 :22	5.95	23 :12	12.29	3 :07	1.65	3 :00	1.38
`j8-per10-2`	1002	1 :03	0.56	8 :46	5.11	8 :50	4.72	1 :17	0.68	1 :13	0.57

TABLE 6.2 – Moyennes du temps moyen de résolution $\bar{t}$ (heure :minute) et du nombre de nœuds $\bar{n}$ (en millions) pour différentes variantes appliquées aux trois instances les plus difficiles.

6.3.2.3 Robustesse

Pour finir, nous analysons la robustesse de RRCP quand on applique la stratégie de Luby avec enregistrement des *nogoods* et que la solution initiale est donnée par CROSH. La robustesse signifie, dans notre cas, la sensibilité à la borne supérieure initiale et aux arbitrages randomisés d'une exécution de RRCP à une autre. Pour chaque instance, nous calculons le quotient de l'écart-type sur la moyenne du temps de résolution : $\frac{\texttt{std}(\bar{t})}{\bar{t}}$. Puis, nous calculons la moyenne de ces quotients pour chaque jeu d'instances. Le jeu d'instances de Taillard obtient le plus haut quotient égal à 62% parce que la preuve d'optimalité est triviale. Les temps de résolution varient énormément en fonction de l'optimalité de la solution initiale d'une exécution à l'autre. En effet, l'instance est résolue sans exploration si la solution initiale est optimale et, dans le cas contraire, le lancement de la recherche arborescente complète entraîne une augmentation du temps de résolution. Au contraire, l'algorithme est plus robuste pour les instances de Guéret-Prins et Brucker où ces quotients sont respectivement égaux à 16% et 9%. En effet, l'algorithme passe plus de temps à prouver l'optimalité de la dernière solution qu'à l'atteindre, car le nombre de nœuds visités pour obtenir la preuve d'optimalité est très élevé.

6.3.3 Comparaison avec d'autres approches

Dans cette section, nous comparons RRCP (sans limite de temps) aux approches de la littérature (voir section 4.3). La variante de RRCP testée est celle utilisant la stratégie de Luby et une solution initiale donnée par CROSH. Les tableaux 6.3, 6.4 et 6.5 récapitulent respectivement les résultats de différentes approches de la littérature sur les jeux d'instances de Taillard, Brucker et Guéret-Prins. Ces tableaux incluent les meilleurs résultats obtenus par un algorithme génétique [GA-Pri – 112], un algorithme de colonies de fourmis [ACO-Blu – 114], un algorithme d'essaim de particules [PSO-Sha – 115], un branch-and-bound avec retour arrière intelligent [BB-Gue – 119], un branch-and-bound avec test de consistance [BB-Do – 121], et une reformulation vers SAT [SAT-Ta – 31]. Les travaux cités ci-dessus mentionnent quelquefois plusieurs résultats obtenus avec différentes variantes de leur approche et nous n'avons gardé que les meilleurs d'entre eux dans les Tableaux 6.3, 6.4 et 6.5. La colonne Opt donne l'optimum pour chaque instance. La valeur de l'objectif est indiquée en gras lorsqu'il est égal à optimum. Plus précisemment :

GA-Pri : nous rapportons la valeur de l'objectif après une exécution unique mais pas le temps d'exécution.

ACO-Blu : les résultats des 20 exécutions pour chaque instance sont obtenus sur des PCs avec un processeur AMD Athlon cadencé à 1.1 Ghz tournant sous Linux. Nous rapportons la valeur moyenne de l'objectif (Avg) et le temps moyen de résolution ($\bar{t}$). La meilleure valeur de l'objectif (Best) obtenue pendant les 20 exécutions est précisée lorsqu'elle n'est pas égale à l'optimum pour toutes les instances.

PSO-Sha : les résultats des 20 exécutions pour chaque instance de PSO-Sha sont obtenus sur des PCs avec un processeur AMD Athlon cadencé à 1.8 Ghz tournant sous Windows XP. Nous rapportons les mêmes informations que pour ACO-Blu. La meilleure valeur de l'objectif de PSO-Sha est précédée du symbole † si l'opérateur de décodage n'est pas hybridé avec une recherche arborescente tronquée.

BB-Gue : nous ne mentionnons pas les temps d'exécution, car la limite de 250 000 backtracks était souvent atteinte (approximativement 3 heures de temps CPU sur un Pentium PC cadencé à 133 MHz).

BB-Do : nous mentionnons les temps de résolution t de BB-Do obtenus sur un Pentium II 333 Mhz tournant sous MSDOS avec une limite de temps de 5 heures car ils ont identifié de nombreuses solutions optimales. Le symbole – indique que leur algorithme bottom-up a été interrompu sans trouver de solution. La meilleure borne supérieure est précisée entre parenthèses lorsque leur algorithme top-down a été interrompu avant d'obtenir la preuve d'optimalité.

SAT-Ta : les temps de résolution de SAT-Ta obtenus sur un Intel Xeon cadencé à 2.8GHz et 4GB de mémoire vive sont donnés à l'exception de ceux des instances `j7-per0-0` et `j8-per0-1` qui ont été résolues en parallèle sur 10 Mac mini (PowerPC G4, 1.42GHz, 1GB de mémoire vive) en divisant chaque problème en 120 sous-problèmes. Les solutions optimales ont été découvertes et prouvées en 13 heures de calcul (noté M dans le tableau 6.4).

CNR-Cha : Chatzikokolakis *et al.* [113] interrompent leur recherche locale après 120 minutes, mais seulement si le temps écoulé depuis la dernière amélioration dépasse 30 minutes. CNR-Cha ne précise ni les temps de résolution, ni les meilleures solutions découvertes. Par conséquent, ces résultats ne sont pas mentionnés dans les tableaux 6.3, 6.4 et 6.5.

MCS-Lab : Laborie [122] exécute sur un portable Dell Latitude D600 cadencé à 1.4 GHz son algorithme bottom-up en imposant une limite de 5 secondes sur la résolution de chaque sous-problème. L'algorithme est interrompu sans retourner de solution dès que la limite de temps est atteinte pour un sous-problème. MCS-Lab ne précise pas les temps de résolution et n'apparaissent pas non plus dans les tableaux. Nous utiliserons, à titre indicatif, une estimation du temps de résolution basée sur les considérations présentés ci-dessous issues de la section 2.4.

Un algorithme bottom-up résout $C_{max} - C^{LB}_{max}$ problèmes insatisfiables et un unique problème satisfiable donnant l'optimum. Ce nombre de sous-problèmes dépend donc de l'instance considérée même si la valeur de C^{LB}_{max} est toujours égale à 1000 pour les jeux d'instances de Brucker et Guéret-Prins. MCS-Lab précise que les premières itérations étaient courtes et que les dernières s'allongeaient durant une transition de phase. Par conséquent, notre estimation du temps de résolution de MCS-Lab est inspirée par la variante dichotomique de bottom-up : $5 \times \lceil \log_2 \left(C_{max} - C^{LB}_{max} + 1\right) + 1 \rceil$ secondes.

Nous rappelons que nos expérimentations ont été menées sur une grille de calcul composée de machines Linux, où chaque nœud possède 1 GB de mémoire vive et un processeur AMD cadencé à 2.2 GHz. Nous rapportons la moyenne sur 20 exécutions du temps de résolution $\bar{t}$ et du nombre de nœuds visités $\bar{n}$. Remarquez que la comparaison des temps de résolution n'est pas toujours significative en raison des différentes configurations matérielles.

Résultats sur les instances de Taillard (Table 6.3) Ce jeu d'instances est réputé facile puisque la preuve d'optimalité est triviale et qu'il est ainsi résolu efficacement par les métaheuristiques. Ainsi, six instances seulement restent ouvertes après une unique exécution de GA-Pri. ACO-Blu et PSO-Sha ferment les dernières instances avec de bons temps de résolution même si certaines exécutions ne découvrent pas l'optimum. Ces échecs ne sont pas clairement reliés à la taille des instances. Au contraire, CNR-Cha obtient ses plus faibles résultats sur ce jeu d'instances puisqu'il ne découvre que huit solutions optimales parmi les instances de taille 7×7 et 10×10, et ne fournit aucun résultat pour les grandes instances. Les temps moyens de résolution de `RRCP` sont comparables à ceux des métaheuristiques sauf pour les instances `tai_20_20_02` et `tai_20_20_08`. Cependant, toutes les métaheuristiques échouent au moins une fois sur l'instance `tai_20_20_08`. On constate aussi que `CROSH` est plus efficace que des métaheuristiques complexes sur de nombreuses grandes instances (si le nombre moyen de nœuds visités $\bar{n}$ est nul, alors toutes les exécutions de `CROSH` découvrent l'optimum).

BB-Do est la première méthode exacte à résoudre toutes les instances 10×10 et la plupart des 15×15 et 20×20. Les performances de leur algorithme bottom-up surpassent clairement celle de leur algorithme top-down sur ce jeu d'instances puisqu'il n'a besoin que de résoudre un unique sous-problème satisfiable pour découvrir l'optimum. Dans notre cas, la précision de la borne supérieure initiale donnée par `CROSH` compense les inconvénients relatifs à l'usage d'un algorithme top-down. De plus, la diversification issue de la randomisation et des redémarrages aide à remettre en cause les mauvais choix initiaux alors que plusieurs instances restent sans solution avec BB-Do, par exemple `tai_15_15_02`. Avec des configurations matérielles équivalentes, les performances de `RRCP` outrepassent clairement celles de SAT-Ta, la première méthode exacte à fermer ce jeu d'instances. Par contre, les temps de résolution de SAT-Ta sont une fonction linéaire du nombre de clauses, donc de la taille des instances, ce qui n'est pas nécessairement le cas pour `RRCP`. Contrairement aux autres approches, SAT-Ta ne résout pas l'instance `tai_20_20_08` plus difficilement que les autres 20×20. Pour finir, MCS-Lab ne rapporte aucun résultat sur le jeu d'instances de Taillard.

Résultats sur les instances de Brucker (Table 6.4) Le jeu d'instances de Brucker est né en réaction à la faible difficulté des instances de Taillard. GA-Pri est l'approche la plus affectée puisqu'elle ne découvre que cinq solutions optimales. Les temps de résolution de ACO-Blu et PSO-Sha augmentent par rapport à ceux sur le jeu d'instances de Taillard et certaines solutions optimales ne sont jamais découvertes. Au contraire, CNR-Cha rapporte l'amélioration de 3 bornes supérieures et la preuve de 17 solutions optimales. `RRCP` obtient de bons temps de résolution par rapport aux métaheuristiques à l'exception de

Instance		Metaheuristics					Exact Algorithms			
		GA-Pri	ACO-Blu		PSO-Sha		BB-Do	SAT-Ta	RRCP	
Name	Opt		Avg	$\bar{t}$	Avg	$\bar{t}$	t	t	$\bar{t}$	$\bar{n}$
tai_7_7_1	435	436	**435**	2.1	**435**	2.9	0.4	21	1.6	355
tai_7_7_2	443	447	**443**	19.2	**443**	12.2	0.9	24	1.6	448
tai_7_7_3	468	472	**468**	16.0	**468**	9.2	30.9	30	4.3	1159
tai_7_7_4	463	**463**	**463**	1.7	**463**	3.0	5.3	20	1.5	478
tai_7_7_5	416	417	**416**	2.3	**416**	2.9	2.0	22	0.8	157
tai_7_7_6	451	455	451.4	24.8	**451**	13.5	95.8	45	11.5	3945
tai_7_7_7	422	426	422.2	23.0	**422**	13.6	167.7	33	2.3	602
tai_7_7_8	424	**424**	**424**	1.2	**424**	2.3	5.0	20	0.6	189
tai_7_7_9	458	**458**	**458**	1.1	**458**	1.3	0.8	21	0.3	109
tai_7_7_10	398	**398**	**398**	1.6	**398**	2.8	53.2	20	0.5	105
tai_10_10_1	637	**637**	637.4	40.1	**637**	9.4	30.2	98	8.3	1214
tai_10_10_2	588	**588**	**588**	3.0	**588**	3.5	70.6	95	4.8	667
tai_10_10_3	598	**598**	**598**	27.9	**598**	10.1	185.5	92	8.5	1162
tai_10_10_4	577	**577**	**577**	2.6	**577**	2.6	29.7	92	2.2	264
tai_10_10_5	640	**640**	**640**	8.6	**640**	4.0	32.0	96	6.6	830
tai_10_10_6	538	**538**	**538**	2.6	**538**	1.1	32.7	95	0.4	0
tai_10_10_7	616	**616**	**616**	5.2	**616**	3.9	30.9	103	4.4	403
tai_10_10_8	595	**595**	**595**	15.0	**595**	7.0	44.1	95	6.0	633
tai_10_10_9	595	**595**	**595**	5.1	**595**	4.1	39.8	97	5.8	541
tai_10_10_10	596	**596**	**596**	7.5	**596**	5.0	29.1	95	5.6	541
tai_15_15_1	937	**937**	**937**	14.3	**937**	4.3	481.4	523	4.4	0
tai_15_15_2	918	**918**	**918**	21.1	**918**	9.1	–	567	26.5	2190
tai_15_15_3	871	**871**	**871**	14.3	**871**	4.3	611.6	543	3.4	0
tai_15_15_4	934	**934**	**934**	14.2	**934**	3.9	570.1	560	1.7	0
tai_15_15_5	946	**946**	**946**	25.7	**946**	5.7	556.3	541	8.5	1760
tai_15_15_6	933	**933**	**933**	16.6	**933**	4.7	574.5	560	3.0	0
tai_15_15_7	891	**891**	**891**	20.1	**891**	10.4	724.6	566	16.5	1896
tai_15_15_8	893	**893**	**893**	14.2	**893**	17.3	614.0	546	1.3	0
tai_15_15_9	899	**899**	899.7	4.1	899.2	26.6	646.9	568	39.2	4053
tai_15_15_10	902	**902**	**902**	18.1	**902**	6.9	720.1	586	22.9	2081
tai_20_20_1	1155	**1155**	**1155**	54.1	**1155**	16.6	3519.8	3105	32.4	3340
tai_20_20_2	1241	**1241**	**1241**	79.7	**1241**	23.5	–	3559	588.4	45606
tai_20_20_3	1257	**1257**	**1257**	48.6	**1257**	19.6	4126.3	2990	3.0	0
tai_20_20_4	1248	**1248**	**1248**	49.1	**1248**	19.6	–	3442	2.7	0
tai_20_20_5	1256	**1256**	**1256**	49.1	**1256**	19.6	3247.3	3603	3.7	0
tai_20_20_6	1204	**1204**	**1204**	49.3	**1204**	19.6	3393.0	2741	10.2	1879
tai_20_20_7	1294	**1294**	**1294**	65.0	**1294**	25.4	2954.8	2912	86.9	8620
tai_20_20_8	1169	1171	1170.3	27.9	1170	50.9	–	2990	305.8	25503
tai_20_20_9	1289	**1289**	**1289**	48.6	**1289**	78.2	3593.8	3204	1.7	0
tai_20_20_10	1241	**1241**	**1241**	48.8	**1241**	78.2	4936.2	3208	1.1	0

TABLE 6.3 – Résultats sur les instances de Taillard.

quelques instances pour lesquelles la comparaison est délicate puisque les temps de résolution de RRCP sont supérieurs, mais les métaheuristiques ne découvrent pas toujours l'optimum.

L'algorithme top-down de BB-Do a prouvé l'optimalité de huit instances 7×7 parmi neuf. MCS-Lab a fermé trois des six dernières instances ouvertes (j8-per0-2, j8-per10-0, and j8-per10-1) et SAT-Ta ferma plus tard les trois dernières (j7-per0-0, j8-per0-1, and j8-per10-2). À l'exception des instances j7-per10-2, j8-per10-0 et j8-per10-1, les temps de résolution de RRCP sont meilleurs que ou similaires aux estimations pour MCS-Lab qui varient entre 5 et 35 secondes. Les temps de résolution de SAT-Ta restent supérieurs à ceux de RRCP, spécialement pour les instances j7-per0-0 et j8-per0-1 pour lesquelles leurs expériences ont demandé un temps de calcul très important. Sur ce jeu d'instances, leurs temps de résolution n'ont plus un comportement linéaire en fonction de la taille du problème.

Résultats sur les instances de Guéret-Prins (Table 6.5) Le jeu d'instance de Guéret-Prins semble difficile puisque les métaheuristiques sont moins efficaces. Les résultats de ACO-Blu et PSO-Sha se dégradent au fur et à mesure que la taille du problème augmente. Malgré des temps d'exécution plus importants, ils découvrent moins de solutions optimales, surtout ACO-Blu. En comparaison, GA-Pri est plus efficace que sur le jeu d'instances de Brucker et CNR-Cha déclare améliorer 12 bornes supérieures. RRCP est particulièrement bien adapté à ce jeu d'instances puisque ses performances surpassent celles des métaheuristiques pour toutes les tailles d'instances.

BB-Gue a prouvé l'optimalité de toutes les instances jusqu'à la taille 6×6 et de quelques instances de taille supérieure. BB-Do ne rapporte aucun résultat sur ce jeu d'instance et MCS-Lab le ferme, mais de

Instance		Metaheuristics							Exact Algorithms			
		GA-Pri	ACO-Blu			PSO-Sha			BB-Do	SAT-Ta	RRCP	
Name	Opt		Best	Avg	$\bar{t}$	Best	Avg	$\bar{t}$	t	t	$\bar{t}$	$\bar{n}$
j6-per0-0	1056	1080	**1056**	**1056**	27.4	**1056**	**1056**	42.1	133.0	817	38.7	11032
j6-per0-1	1045	**1045**	**1045**	1049.7	61.3	**1045**	**1045**	59.7	5.2	57	0.3	198
j6-per0-2	1063	1079	**1063**	**1063**	38.8	**1063**	**1063**	72.6	18.0	57	0.6	223
j6-per10-0	1005	1016	**1005**	**1005**	10.6	**1005**	**1005**	45.5	14.4	52	0.8	263
j6-per10-1	1021	1036	**1021**	**1021**	11.3	**1021**	**1021**	21.0	4.6	46	0.3	177
j6-per10-2	1012	**1012**	**1012**	**1012**	1.4	**1012**	**1012**	8.5	13.8	51	0.5	188
j6-per20-0	1000	1018	**1000**	1003.6	31.1	**1000**	**1000**	77.5	10.7	60	0.4	208
j6-per20-1	1000	**1000**	**1000**	**1000**	0.8	**1000**	**1000**	1.5	0.4	46	0.2	161
j6-per20-2	1000	1001	**1000**	**1000**	3.9	**1000**	**1000**	30.6	1.0	40	0.4	179
j7-per0-0	1048	1071	**1048**	1052.7	207.9	1050	1051.2	104.9	(1058)	M	7777.2	1564192
j7-per0-1	1055	1076	1057	1057.8	91.6	**†1055**	1058.8	155.8	9421.8	428	16.5	3265
j7-per0-2	1056	1082	1058	1059	175.9	**1056**	1057	124.5	9273.5	292	16.4	3120
j7-per10-0	1013	1036	**1013**	1016.7	217.6	**1013**	1016.1	183.8	2781.9	332	19.1	3981
j7-per10-1	1000	1010	**1000**	1002.5	189.9	**1000**	**1000**	81.9	1563.0	121	6.4	1276
j7-per10-2	1011	1035	1016	1019.4	180.7	1013	1014.9	125.6	15625.1	1786	583.1	128289
j7-per20-0	1000	**1000**	**1000**	**1000**	0.4	**1000**	**1000**	1.9	48.8	66	0.1	0
j7-per20-1	1005	1030	**1005**	1007.6	259.1	1007	1008	143.2	318.8	132	8.9	2130
j7-per20-2	1003	1020	**1003**	1007.3	257.3	**1003**	1004.7	160.9	2184.9	132	13.8	3150
j8-per0-1	1039	1075	**1039**	1048.7	313.5	**1039**	1043.3	220.8		M	11168.9	1648700
j8-per0-2	1052	1073	**1052**	1057.1	323.4	**1052**	1053.6	271.9		870	61.3	9379
j8-per10-0	1017	1053	1020	1026.9	346.5	1020	1026.1	205.0		2107	184.5	24548
j8-per10-1	1000	1029	1004	1012.4	308.9	1002	1007.6	202.2		8346	1099.3	165875
j8-per10-2	1002	1027	1009	1013.7	399.4	**1002**	1006	162.8		7789	4596.5	673451
j8-per20-0	1000	1015	**1000**	1001	237.2	**1000**	1000.6	136.9		148	9.1	2104
j8-per20-1	1000	**1000**	**1000**	**1000**	2.6	**1000**	**1000**	4.5		136	0.4	128
j8-per20-2	1000	1014	**1000**	1000.6	286.2	**1000**	**1000**	105.8		144	6.7	1512

TABLE 6.4 – Résultats sur les instances de Brucker.

nouveau sans préciser les temps de résolution. Les temps de résolution de RRCP sont strictement inférieurs aux estimations pour MCS-Lab qui varient entre 40 et 50 secondes. Les temps de résolution de SAT-Ta sont de nouveau une fonction de la taille des instances mais restent peu compétitifs par rapport à ceux de RRCP.

6.4 Conclusion

Nous avons présenté une approche en ordonnancement sous contraintes pour résoudre le problème d'open-shop. L'algorithme s'appuie sur le calcul d'une solution initiale de manière heuristique avant de résoudre un modèle déclaratif basé sur un langage de haut niveau (tâches, ressources, précédences) grâce à une recherche arborescente de type top-down complétée par un mécanisme de randomisation et de redémarrage.
Les résultats expérimentaux sont comparables à ceux des métaheuristiques sur le jeu d'instances de Taillard. De plus, RRCP obtient de meilleures solutions que les métaheuristiques en un temps moindre sur les jeux d'instances de Brucker et Guéret-Prins. Les résultats expérimentaux surpassent ceux des méthodes exactes pour lesquelles les temps de résolution des instances sont disponibles.
Au chapitre 7, nous proposerons et étudierons de nouvelles heuristiques de sélection pour l'arbitrage du graphe disjonctif et nous appliquerons notre algorithme aux problèmes d'open-shop et de job-shop.

Instance		Metaheuristics							Exact Algorithms			
		GA-Pri	ACO-Blu			PSO-Sha			BB-Gue	SAT-Ta	RRCP	
Name	Opt		Best	Avg	$\bar{t}$	Best	Avg	$\bar{t}$		t	$\bar{t}$	$\bar{n}$
gp06-01	1264	**1264**	**1264**	1264.7	30.8	**1264**	**1264**	176.1	**1264**	57	0.3	80
gp06-02	1285	**1285**	**1285**	1285.7	48.7	**1285**	**1285**	147.8	**1285**	65	0.2	172
gp06-03	1255	**1255**	**1255**	**1255**	30.0	**1255**	1255.6	133.1	**1255**	72	0.1	124
gp06-04	1275	**1275**	**1275**	**1275**	25.9	**1275**	**1275**	60.8	**1275**	63	0.1	67
gp06-05	1299	1300	**1299**	1299.2	39.9	**1299**	**1299**	159.6	**1299**	65	0.1	67
gp06-06	1284	**1284**	**1284**	**1284**	43	**1284**	**1284**	109.4	**1284**	65	0.1	68
gp06-07	1290	**1290**	**1290**	**1290**	10.5	**1290**	**1290**	1.6	**1290**	77	0.1	63
gp06-08	1265	1266	**1265**	1265.2	71.9	**1265**	1265.5	134.3	**1265**	71	0.1	52
gp06-09	1243	1243	**1243**	**1243**	9.8	**1243**	1243.1	156.5	1264	72	0.2	170
gp06-10	1254	**1254**	**1254**	**1254**	4.6	**1254**	**1254**	79.8	**1254**	57	0.3	241
gp07-01	1159	**1159**	**1159**	**1159**	86.9	**1159**	1159.3	223.7	1160	99	0.9	367
gp07-02	1185	**1185**	**1185**	**1185**	80.3	**1185**	**1185**	1.2	1191	148	0.6	4
gp07-03	1237	**1237**	**1237**	**1237**	40.9	**1237**	**1237**	9.5	1242	132	0.7	54
gp07-04	1167	**1167**	**1167**	**1167**	59.2	**1167**	**1167**	160.4	1167	131	0.7	144
gp07-05	1157	**1157**	**1157**	**1157**	124.4	**1157**	**1157**	139.1	1191	141	0.8	304
gp07-06	1193	**1193**	**1193**	1193.9	152.4	**1193**	1193.1	198.6	1200	127	0.8	306
gp07-07	1185	**1185**	**1185**	1185.1	91.1	**1185**	**1185**	1.4	1201	102	0.6	48
gp07-08	1180	1181	**1180**	1181.4	206.7	**1180**	**1180**	139.4	1183	144	0.7	117
gp07-09	1220	**1220**	**1220**	1220.1	127.9	**1220**	**1220**	143.9	**1220**	150	0.7	177
gp07-10	1270	**1270**	**1270**	1270.1	65.6	**1270**	**1270**	0.5	**1270**	127	0.6	4
gp08-01	1130	1160	**1130**	1132.4	335.0	**†1130**	1140.3	277.3	1195	160	2.6	1485
gp08-02	1135	1136	**1135**	1136.1	228.4	**1135**	1135.4	258.3	1197	190	1.2	304
gp08-03	1110	1111	1111	1113.7	336.3	**1110**	1114	240.3	1158	197	1.6	622
gp08-04	1153	1168	1154	1156	275.7	**1153**	1153.2	308.1	1168	227	1.4	566
gp08-05	1218	**1218**	1219	1219.8	347.7	**1218**	1218.9	56.6	**1218**	247	1.2	206
gp08-06	1115	1128	1116	1123.2	359.2	**1115**	1126.9	249.6	1171	175	2.3	1498
gp08-07	1126	1128	**1126**	1134.6	296.8	**1126**	1129.8	287.3	1157	204	3.6	2775
gp08-08	1148	**1148**	**1148**	1149	277.4	**1148**	**1148**	179.3	1191	183	2.0	1281
gp08-09	1114	1120	1117	1119	279.0	**1114**	1114.3	223.6	1142	189	2.0	1140
gp08-10	1161	**1161**	**1161**	1161.5	281.3	**1161**	1161.4	217.1	**1161**	203	1.1	245
gp09-01	1129	1143	1135	1142.8	412.9	**1129**	1133.2	376.3	1150	323	3.6	1691
gp09-02	1110	1114	1112	1113.7	430.8	**†1110**	1114.1	335.9	1226	327	10.7	8000
gp09-03	1115	1118	1118	1120.4	428.0	†1116	1117	313.4	1150	395	2.8	1422
gp09-04	1130	1131	**1130**	1140	549.7	**1130**	1135.8	328.7	1181	340	4.3	2219
gp09-05	1180	**1180**	**1180**	1180.5	295.9	**1180**	**1180**	22.3	**1180**	362	1.7	266
gp09-06	1093	1117	**1093**	1195.6	387.0	**1093**	1094.1	277.2	1136	401	4.6	2387
gp09-07	1090	1119	1097	1101.4	431.4	1091	1096.5	376.4	1173	339	5.9	3483
gp09-08	1105	1110	1106	1113.7	376.2	1108	1108.3	334.6	1193	349	3.1	1446
gp09-09	1123	1132	1127	1132.5	402.6	**†1123**	1126.5	358.6	1218	316	3.2	1537
gp09-10	1110	1130	1120	1126.3	435.8	†1112	1126.5	297.7	1166	355	6.1	2784
gp10-01	1093	1113	1099	1109	567.5	**1093**	1096.8	455.7	1151	470	29.8	6661
gp10-02	1097	1120	1101	1107.4	501.7	**1097**	1099.1	382.7	1178	526	9.7	3140
gp10-03	1081	1101	1082	1098	658.7	**†1081**	1090.3	450.8	1162	535	13.6	4196
gp10-04	1077	1090	1093	1096.6	588.1	1083	1092.1	371.8	1165	515	12.4	3921
gp10-05	1071	1094	1083	1092.4	636.4	†1073	1092.2	314.1	1125	515	16.3	4782
gp10-06	1071	1074	1088	1104.6	595.5	**1071**	1074.3	289.7	1179	508	12.4	3894
gp10-07	1079	1083	1084	1091.5	389.6	†1080	1081.1	167.4	1172	523	8.7	2188
gp10-08	1093	1098	1099	1104.8	615.9	†1095	1097.6	324.5	1181	498	10.5	3477
gp10-09	1112	1121	1121	1128.7	554.5	†1115	1127	428.2	1188	541	10.1	3303
gp10-10	1092	1095	1097	1106.7	562.5	**1092**	1094	487.9	1172	656	7.4	1724

TABLE 6.5 – Résultats sur les instances de Guéret-Prins.

Chapitre 7

Heuristiques d'arbitrage dans un atelier

Ce chapitre décrit plusieurs variantes de la méthode de résolution des problèmes d'atelier présentée au chapitre 6. La majorité des approches complètes en ordonnancement sous contraintes sont basées sur des algorithmes de filtrage et des heuristiques de sélection dédiés à ces problèmes. En effet, une croyance répandue est que ces problèmes sont trop difficiles pour être résolus autrement. Nous proposons une nouvelle heuristique inspirée de celle du degré pondéré combinée à un modèle du graphe disjonctif dont la propagation reste naïve. Nous montrons que les résultats de cette approche surpassent souvent ceux des meilleures approches en ordonnancement sous contraintes pour l'open-shop et le job-shop.

Sommaire

7.1 Modèles en ordonnancement sous contraintes 70
7.1.1 Contraintes temporelles 70
7.1.2 Stratégie de branchement 70
7.1.3 Notes sur les modèles 71
7.2 Algorithme de résolution 72
7.3 Évaluations expérimentales 72
7.3.1 Problèmes d'open-shop 72
7.3.2 Problèmes de job-shop 74
7.4 Conclusion 75

Nous présentons nos travaux [5] sur les heuristiques d'arbitrage pour la résolution des problèmes d'atelier par l'algorithme présenté au chapitre 6. Nous examinerons la variation des performances des heuristiques d'arbitrage (bloc 9 de l'algorithme en figure 6.2) en fonction de différents modèles en ordonnancement sous contraintes (bloc 3). Des expérimentations préliminaires, non détaillées ici, ont confirmé les résultats de l'analyse de sensibilité présentée en section 6.3.2 page 60. De ce fait, nous ne remettrons pas en cause les autres composantes de l'algorithme.

En constatant l'importance des redémarrages, nous avons souhaité exploiter les informations issues des différentes exécutions autrement que par l'enregistrement des *nogoods*. L'heuristique du degré pondéré (voir section 2.3) a attiré notre attention, car il s'agit d'une alternative à la randomisation de l'arbitrage (bloc 7). En effet, il suffit de conserver les poids des contraintes d'une exécution à l'autre pour espérer résoudre plus facilement le problème en identifiant les parties difficiles (*backdoors*) qui correspondent aux variables dont le degré pondéré est élevé. Par conséquent, la stratégie de branchement doit maintenant instancier une variable à chaque point de choix et non ajouter une contrainte. Pour ce faire, nous délaisserons la contrainte globale modélisant le graphe de précédence au profit d'un ensemble de contraintes de disjonction réifiées modélisant le graphe disjonctif. Ces changements nous amèneront à distinguer deux modèles en fonction de la présence de contraintes disjonctives.

La section 7.1 décrit les modèles *Light* et *Heavy* puis les nouvelles stratégies de branchement. Nous uti-

lisons deux implémentations du modèle *Light* basées sur les solveurs `choco` et `mistral` (voir section 2.5). La section 7.2 présente les différences mineures entre les implémentations `choco` et `mistral`. Finalement, La section 7.3 démontre la pertinence de cette approche sur les problèmes d'open-shop et de job-shop. Nous croyons que la combinaison de l'heuristique inspirée du degré pondéré et du modèle *Light* est très efficace pour identifier les paires de tâches critiques.

7.1 Modèles en ordonnancement sous contraintes

Nous présentons dans cette section deux modèles logiquement équivalents, mais dont les niveaux d'inférence sont différents. Ces modèles sont basés sur la modélisation du graphe disjonctif par des contraintes de disjonction réifiées présentée en section 7.1.1 qui remplace la contrainte globale décrite en section 6.1.2. Ils sont destinés à être combinés avec les stratégies de branchement présentées en section 7.1.2. Le modèle *Heavy* pose des contraintes globales disjonctives définies en section 6.1 contrairement au second modèle, dénommé *Light*. Finalement, la section 7.1.3 discute de ces deux modèles et de la modélisation des contraintes temporelles dans le chapitre 6 et dans celui-ci.

7.1.1 Contraintes temporelles

Ainsi, nous passons d'une représentation du graphe de précédences par une contrainte globale à une représentation du graphe disjonctif par des contraintes de disjonction entre deux tâches définies ci-dessous. Nous introduisons une variable booléenne $b_{ij \preceq kl}$ qui représente l'ordre d'enchaînement pour chaque paire de tâches T_{ij} et T_{kl} partageant une ressource (lot ou machine). La variable booléenne $b_{ij \preceq kl}$ réifie la contrainte de disjonction présentée en section 3.2 en prenant la valeur 1 lorsque T_{ij} précède T_{kl} et la valeur 0 lorsque T_{kl} précède T_{ij} :

$$T_{ij} \simeq T_{kl} \quad \Leftrightarrow \quad \begin{cases} T_{ij} \preceq T_{kl} & \text{si } b_{ij \preceq kl} = 1 \\ T_{kl} \preceq T_{ij} & \text{si } b_{ij \preceq kl} = 0 \end{cases} \tag{7.1}$$

Tant que la disjonction n'est pas arbitrée, la contrainte applique les règles de filtrage : précédence interdite - précédence obligatoire (voir section 3.2). Après son arbitrage par l'instanciation de la variable booléenne $b_{ij \preceq kl}$, le filtrage applique une règle assurant la consistance de bornes de la contrainte de précédence. Pour n lots et m machines, la représentation du graphe disjonctif nécessite $\frac{nm(m+n-2)}{2}$ disjonctions réifiées pour l'open-shop et $\frac{nm(m+n-2)}{4}$ pour le flow-shop ou le job-shop. Dans le cas du flow-shop et du job-shop, l'ordre des opérations dans un lot est imposé par l'ajout des contraintes de précédences (arithmétiques) entre les paires d'opérations consécutives.

7.1.2 Stratégie de branchement

Un arbitrage complet du graphe disjonctif (impliquant l'existence d'une solution) est maintenant obtenu en instanciant toutes les variables booléennes réifiant les disjonctions entre les paires de tâches partageant une ressource (lot ou machine).

Sauf mention contraire, on sélectionne aléatoirement (distribution uniforme) une des meilleures variables en cas d'égalité.

profile instancie maintenant une variable booléenne à chaque point de choix au lieu d'ajouter une contrainte de précédence (voir section 6.1.4).

top10%-profile sélectionne aléatoirement (distribution uniforme) une disjonction appartenant au 10 % des meilleures paires (ressource/instant) des profils.

slack sélectionne une variable $b_{ij \preceq kl}$ dont le « domaine » est de taille minimale. La taille du domaine est mesurée comme la somme des tailles des fenêtres de temps dans la disjonction $T_{ij} \simeq T_{kl}$: $(lst_{ij} - est_{ij} + 1) + (lst_{kl} - est_{kl} + 1)$.

wdeg sélectionne la variable dont le poids pondéré de ses contraintes est maximal. Le poids d'une contrainte est initialisé à la valeur de son degré. Il est incrémenté à chaque fois qu'elle déclenche une contradiction pendant la propagation. Dans notre modèle, toutes les variables de décision $b_{ij \preceq kl}$

sont booléennes et *wdeg* est par conséquent équivalente à *dom/wdeg*. Le degré des variables $b_{ij \preceq kl}$ est unitaire.

slack/wdeg sélectionne une variable $b_{ij \preceq kl}$ dont le quotient du « domaine » sur le degré pondéré est minimal.

top3-slack, top3-wdeg, top3-dom/wdeg sélectionnent aléatoirement (distribution uniforme) une des trois meilleures variables quand il n'y a pas d'égalité. En cas d'égalité, elles sélectionnent aléatoirement une des meilleures variables

L'heuristique *wdeg* ne devient discriminante que lorsqu'une première contradiction a été levée. Ainsi, l'heuristique *slack/wdeg* comporte un élément de discrimination basé sur les fenêtres de temps jusqu'à l'entrée en lice des degrés pondérés.

Nous considérons trois heuristiques de sélection de valeur (ou arbitrage) :

minVal sélectionne la plus petite valeur du domaine.

centroid sélectionne la valeur associée à la précédence qui préserve l'ordre des centroids (voir section 6.1.4). En cas d'égalité, on sélectionne aléatoirement (distribution uniforme) une précédence.

jobOrdering est une heuristique statique pour le job-shop et le flow-shop où une paire d'opérations est ordonnée sur une machine selon leur position relative dans leur lot respectif. Par exemple, si la tâche T_{ij} est la seconde de son lot et T_{ik} est la quatrième, l'heuristique sélectionne la valeur 1 pour $b_{ij \preceq ik}$, c'est-à-dire T_{ij} précède T_{ik}.

Comme précédemment, *centroid* est toujours combiné à *profile*. Sauf mention contraire, *minVal* est combiné aux autres heuristiques

Le solveur `choco` propose maintenant plusieurs heuristiques d'arbitrages basées sur la fonction proposée par Laborie [122] : $preserved(x \leq y)$ (voir chapitre 9). Ces dernières n'ont pas été évaluées pendant cette thèse.

7.1.3 Notes sur les modèles

L'heuristique du degré pondéré est très sensible à la modélisation puisque deux ensembles de contraintes logiquement équivalents provoquent des répartitions différentes des poids. En effet, les poids dépendent des contraintes du modèle, mais aussi de l'ordre de propagation des contraintes par le solveur. Lorsqu'une inconsistance peut être détectée par plusieurs contraintes, l'ordre de propagation des contraintes détermine quel poids va être incrémenté. La présence des contraintes disjonctives perturbe donc l'apprentissage des poids si elles sont propagées avant les disjonctions. Par contre, les réductions supplémentaires des fenêtres de temps augmentent la précision du profil des ressources ce qui peut bénéficier à *profile*.

La première qualité de la contrainte globale associée au graphe de précédence du chapitre 6 est d'offrir une propagation optimisée et incrémentale sur le sous-réseau de contraintes. Ensuite, le branchement et le filtrage peuvent exploiter l'information structurelle à propos du graphe de précédences (fermeture transitive, ordre topologique). Enfin, on ne réalise aucune hypothèse sur l'ensemble des disjonctions $\mathcal{D}$ du problème ce qui permet de traiter éventuellement des problèmes dynamiques ou cumulatifs.
En revanche, les contraintes sont ajoutées de manière réversible et toute information associée est perdue lors d'un retour arrière. L'apprentissage des poids des disjonctions est donc compliqué d'autant plus qu'il faudrait expliquer les échecs provoqués par la contrainte globale. Par ailleurs, la gestion des temps d'attente dans un graphe disjonctif généralisé n'est pas immédiate puisqu'elle modifie la relation de transitivité.

Les contraintes de disjonctions réifiées représentant le graphe disjonctif sont taillées sur mesure pour l'apprentissage des poids. L'introduction des variables booléennes $b_{ij \preceq kl}$ permet d'imposer facilement des restrictions supplémentaires sur les gammes opératoires par l'intermédiaire de contraintes booléennes. Elles facilitent aussi l'enregistrement des *nogoods* qui sont toujours définis pour une borne supérieure courante *ub* mais par un ensemble d'instanciations de variables booléennes $b_{ij \preceq kl}$ au lieu d'un ensemble de contraintes de précédence P. Un *nogood* est alors défini par une clause booléenne dont la propagation peut être déléguée au solveur et bénéficier ainsi d'éventuelles fonctionnalités dédiées au problème SAT. De plus, la gestion du graphe disjonctif généralisé demande uniquement une modification mineure de l'implémentation des contraintes de disjonction. Finalement, nous verrons qu'une propagation naïve permet d'explorer rapidement un nombre conséquent de nœuds améliorant ainsi l'apprentissage des poids.
L'inconvénient majeur est que ce modèle implique de fixer l'ensemble des disjonctions $\mathcal{D}$ avant le début de

la recherche. Ensuite, la propagation des précédences est déléguée au solveur et il devient alors difficile de l'optimiser puisqu'elle dépend de l'ordre dans lequel les contraintes sont propagées. La perte de l'information structurelle complique la détection de la transitivité et des cycles. Pour pallier cet inconvénient, on peut envisager d'implémenter une nouvelle contrainte globale structurelle (ne propageant pas les fenêtres de temps) ou de poser des clauses booléennes associées aux inégalités triangulaires entre les précédences (voir section 9.4). Dans ce chapitre, nous n'appliquons aucune de ces alternatives.

7.2 Algorithme de résolution

La solution initiale est calculée en `choco` par `CROSH`, mais par une variante incomplète de dichotomic-bottom-up (voir section 2.4) en `mistral` pour cause d'incompatibilités de langages (Java vs C). Dans cette variante, nous imposons une limite de temps à chaque étape.
Dans toutes les évaluations, les *nogoods* sont enregistrés lors des redémarrages. Les *nogoods* sont maintenant définis par des clauses booléennes. La contrainte globale propageant les clauses n'incrémente son poids lors de la détection d'une inconsistance. En effet, cette information est non discriminante puisque toutes les variables $b_{ij \preceq kl}$ appartiennent à cette contrainte globale. De surcroît, cette augmentation peut influencer les décisions ultérieures, par exemple lorsque les variables de décision ne sont pas toutes booléennes.

7.3 Évaluations expérimentales

Toutes les expérimentations ont été menées sur une grille de calcul composée d'Intel Xeon cadencés à 2.66GHz (12GB de mémoire vive) tournant sous Fedora 9. La configuration matérielle est plus puissante que dans le chapitre 6. Les évaluations concernent les problèmes d'open-shop (section 7.3.1) et de job-shop (section 7.3.2). La durée d'une exécution de l'algorithme est limitée à 3600 secondes. Les évaluations apportent la preuve qu'il est possible de résoudre ces problèmes d'atelier sans utiliser de techniques complexes d'inférence et de branchement. Nous avons réalisé différentes expérimentations dans le but de : (a) comparer les différents modèles (*Light*, *Heavy*) et heuristiques (*profile*, *slack/wdeg*) pour l'open-shop et le job-shop ; (b) évaluer l'importance des deux composantes de l'heuristique *slack/wdeg*.
Le solveur `choco` permettra une comparaison poussée des différentes configurations sur l'open-shop. Nous utiliserons aussi une implémentation du modèle *Light* avec le solveur `mistral` [27] dont nous verrons que la rapidité est un atout majeur pour la résolution de ce modèle. Ensuite, nous comparerons le modèle *Light* en `mistral` avec un modèle voisin du *Heavy* implémenté avec `Ilog Scheduler` (aimablement fourni par Beck [123].

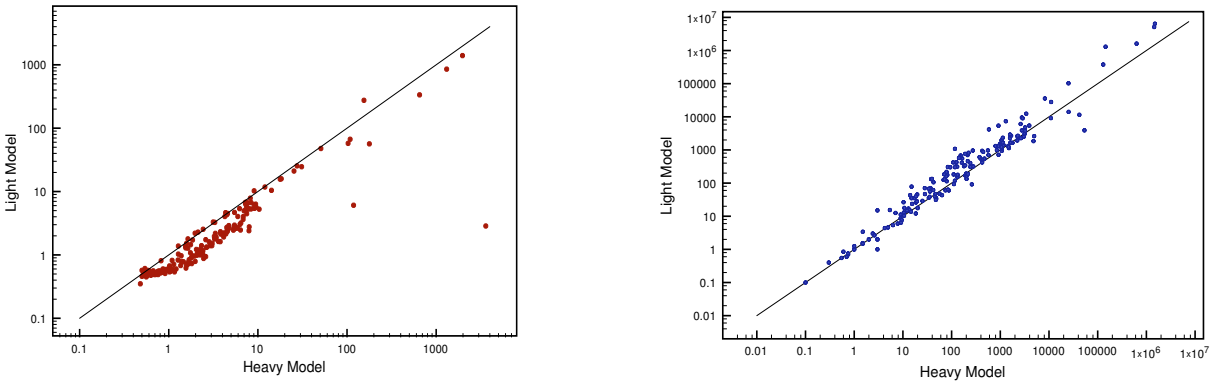
7.3.1 Problèmes d'open-shop

Nous utilisons les jeux d'instances de la littérature Taillard, Brucker et Guéret-Prins composés de 192 instances dont la taille varie entre 3×3 et 20×20 (voir section 6.3).

7.3.1.1 Comparaison des modèles *Light* et *Heavy*

Nous comparons 4 modèles implémentés en `choco`, les modèles classiques *Light* et *Heavy* (« *Light-slack/wdeg* » et « *Heavy-profile* »), ainsi que les variantes où les heuristiques sont inversées (*Light-profile* et *Heavy-slack/wdeg*). Le modèle *Light* en `mistral` se différencie de celui de `choco` lors du calcul de la solution initiale par une procédure d'optimisation dichotomique à la place de `CROSH`. Pour comparer les deux solveurs, le modèle *Light* utilise la politique de redémarrage de Walsh où les facteurs d'échelle et géométrique sont respectivement égaux à 256 et 1.3. Tous les autres modèles utilisent les réglages finaux du chapitre 6. Les modèles sont résolus 20 fois avec une limite de temps de 3600 secondes. Le générateur de nombre aléatoire utilise une graine différente à chaque exécution.

La figure 7.1 compare la résolution des 192 instances d'open-shop par les modèles *Light* et *Heavy*. Chaque point représente une instance et ses coordonnées x et y sont respectivement les temps de résolution (figure 7.1(a)) et les nombres de nœuds (figure 7.1(a)) du modèle *Heavy* et *Light*. Les échelles des axes sont logarithmiques.



(a) Temps de résolution.

(b) Nombre de nœuds.

FIGURE 7.1 – Comparaison des modèles *Light* et *Heavy* sur le problème d'open-shop.

Le tableau 7.1 récapitule les résultats sur les 11 instances les plus difficiles, c'est-à-dire pour lesquelles le temps moyen de résolution est supérieur à 20 secondes pour au moins une approche. Cette sélection est constituée de 1 instance de Guéret-Prins, 4 de Taillard et 6 de Brucker dont la taille varie entre 7×7 et 20×20. Nous rapportons les temps moyens de résolution (colonne $\bar{t}$) en secondes et nombre moyen de nœuds visités (colonne $\bar{n}$) en milliers (K) ou millions (M). Les moyennes intègrent les résultats des exécutions pour lesquelles le modèle n'a pas prouvé l'optimalité dans la limite de temps (3600 secondes). Ces moyennes sont donc en fait des bornes inférieures. Les champs en gras sont les minima sur l'ensemble des résultats.

	Light-slack/wdeg		*Heavy-profile*		*Light-profile*		*Heavy-slack/wdeg*		`mistral`	
Name	$\bar{t}$	$\bar{n}$	$\bar{t}$	$\bar{n}$	$\bar{t}$	$\bar{n}$	$\bar{t}$	$\bar{n}$	$\bar{t}$	$\bar{n}$
`GP10-01`	6.1	4K	118.4	53K	2523.2	6131K	9.6	**3K**	**0.3**	**3K**
`j7-per0-0`	854.5	5.1M	1310.9	**1.4M**	979.1	4.3M	3326.7	2.6M	**327.0**	8.7M
`j7-per10-2`	57.6	0.4M	102.7	**0.1M**	89.5	0.4M	109.6	**0.1M**	**33.5**	1.2M
`j8-per0-1`	1397.1	6.4M	1973.8	**1.5M**	1729.3.	6.0M	3600.0	2.4M	**427.1**	10M
`j8-per10-0`	24.6	0.10M	30.9	**0.02M**	19.7	0.05M	68.0	0.06M	**17.6**	0.47M
`j8-per10-1`	275.2	1.3M	154.8	**0.1M**	92.8	0.3M	796.7	0.7M	**89.3**	2.3M
`j8-per10-2`	335.3	1.6M	651.4	**0.6M**	754.0	2.7M	697.1	**0.6M**	**93.1**	2.3M
`tai-20-1`	25.4	**2K**	27.5	3K	2524.7	1458K	34.4	3K	**3.9**	21K
`tai-20-2`	56.5	11K	178	42K	3600.0	2261K	81.9	**10K**	**14.1**	61K
`tai-20-7`	47.8	9K	60.9	11K	3600.0	2083K	63.7	**8K**	**9.5**	47K
`tai-20-8`	66.8	14K	108.3	25K	3600.0	2127K	84.8	**11K**	**8.2**	39K
Total Avg	286.1	1.3M	428.9	**0.4M**	1774.3	2.5M	806.6	0.6M	**93.0**	2.3M

TABLE 7.1 – Temps de résolution des 11 instances les plus difficiles du problème d'open-shop.

Les modèles *Light* et *Heavy* en `choco` résolvent toutes les instances. Par contre, le modèle *Light* est généralement plus rapide, notamment sur 10 des 11 instances les plus difficiles, malgré l'exploration d'un nombre bien plus important de nœuds. Le modèle *Light* en `mistral` est le plus rapide. Le nombre de nœuds de `mistral` est majoré, car il comprend la phase de recherche dichotomique. Il est intéressant de remarquer que sur les six instances difficiles de Brucker, le modèle *Heavy* est légèrement plus rapide que le modèle *Light* (`choco`) pour découvrir une solution optimale (310s vs 360s), mais est trois fois plus lent pour prouver son optimalité (385s vs 125s). Cela confirme notre intuition sur la capacité de l'heuristique du degré pondéré à identifier les paires de tâches critiques.

Dans l'espoir de mieux comprendre le lien entre les heuristiques et les modèles, nous examinons les variantes où les heuristiques sont inversées ce qui a entraîné dans les deux cas une dégradation des performances. Pour le modèle *Light*, *slack/wdeg* est significativement plus efficace que *profile*. En effet, la variante *Light-profile* ne prouve l'optimalité que dans 60 % des exécutions et leur temps moyen augmente

quasiment d'un ordre de grandeur par rapport à celui de *Light-slack/wdeg*. Pour le modèle *Heavy*, la situation s'inverse puisque *slack/wdeg* est en moyenne plus lent que *profile* même si le taux d'exécution optimale de *slack/wdeg* atteint 89 %. En effet, wdeg est moins discriminante dans le modèle *Heavy* où les nombreux échecs provoqués par les contraintes disjonctives (leurs inférences sont plus fortes) ne sont pas discriminants.

Finalement, des expérimentations supplémentaires sur l'utilisation de la stratégie de Luby avec le modèle *Light* ont également montré une dégradation des performances. Au contraire, nous avions montré que les réglages finaux du modèle *Heavy* obtenaient des performances équivalentes au chapitre 6.

7.3.1.2 Influence des composantes de slack/wdeg

Nous évaluons maintenant l'influence des deux composantes (taille des fenêtres de temps et degré pondéré de notre heuristique *slack/wdeg* sur la résolution du modèle *Light*. Pour cela, nous comparons la résolution du modèle *Light* de `mistral` par les heuristiques *top3-slack*, *top3-wdeg* et *slack/wdeg*. L'élément de randomisation supplémentaire des heuristiques *top3-slack* et *top3-wdeg* améliore la diversification au début de la recherche. Chaque étape de la procédure dichotomique est limitée à 30 secondes pour permettre à toutes les heuristiques de démarrer avec une bonne solution initiale.

	top3-slack		*top3-wdeg*		*slack/wdeg*	
Name	%	$\bar{t}$	%	$\bar{t}$	%	$\bar{t}$
Brucker	89.2	1529.2	100.0	381.4	100.0	249.7
Taillard	0.0	3600.0	93.7	1423.9	100.0	8.8
Total Avg	53.0	2358.5	97.0	798.4	100.0	153.3

TABLE 7.2 – Comparaison des heuristiques de sélection de disjonction sur le problème d'open-shop.

Le tableau 7.2 récapitule les pourcentages d'exécution prouvant la solution optimale (colonnes %) et les temps moyens de résolution (colonnes $\bar{t}$) pour 20 exécutions sur chaque instance. L'heuristique *top3-slack* est relativement peu efficace sur ces instances puisqu'elles ne prouvent l'optimum que pour 53 % des exécutions et que les bornes supérieures sont faibles pour les instances de Taillard.
Les résultats montrent clairement l'efficacité de *top3-wdeg* sur ces instances puisque l'optimum est prouvé pour 97 % des exécutions et la borne supérieure est à distance au plus unitaire de l'optimum en cas d'échec. Par contre, *top3-wdeg* ne discrimine pas les variables au début de la recherche puisque les degrés pondérés sont unitaires jusqu'à ce qu'une première contradiction soit levée. Cette première contradiction apparaît quelquefois profondément dans l'arbre de recherche selon la taille du problème, la borne supérieure initiale, la charge de travail des lots et machines. En fait,la sélection des disjonctions est aléatoire avant la première contradiction.
Finalement, la combinaison de l'information sur les domaines et les degrés pondérés dans *slack/wdeg* augmente le taux de résolution à 100 % tout en diminuant les temps de résolution. L'information sur les domaines : (a) discrimine les disjonctions au début de la recherche ; (b) améliore la qualité des premiers poids appris en privilégiant les disjonctions dont les arbitrages ont une probabilité élevée d'être inconsistants. Des expériences préliminaires, non détaillées ici, ont montré que la randomisation des x meilleurs choix dégrade les performances de *slack/wdeg* même quand x est petit. Cette dégradation s'accentue quand x augmente. Contrairement aux autres heuristiques, la randomisation de la sélection d'une disjonction est moins importante pour les heuristiques du degré pondéré, car l'apprentissage de la pondération joue le rôle d'un processus de diversification.

7.3.2 Problèmes de job-shop

Nous comparons maintenant le modèle *Light* en `mistral` à l'algorithme nommé *randomized restart* utilisé par Beck [123] et implémenté avec `Ilog Scheduler`. Il est important de remarquer que nous ne comparons pas à l'algorithme *solution guided multi point constructive search* (SGMPCS), mais à l'algorithme *randomized restart* qui est utilisé à titre de comparaison dans ses expérimentations. Nos expérimentations utilisent la configuration matérielle décrite plus haut et la version 6.2 d'`Ilog Scheduler`.

Toutes les valeurs des paramètres sont prises de [123]. La politique de redémarrage est donc celle de Luby sans facteur d'échelle ($s = 1$). Un élément de randomisation supplémentaire est ajouté en utilisant l'heuristique de sélection de disjonction *top10%-profile*. Finalement, plusieurs techniques d'inférence forte sont intégrées au modèle telles que les techniques de *time-table* [167], d'*edge finding* [168] et de la contrainte *balance* [169].
Les paramètres de `mistral` sont identiques à ceux de l'open-shop à l'exception d'une limite de 300 secondes pour chaque étape de la recherche dichotomique. Nous utilisons aussi l'heuristique d'arbitrage *jobOrdering* dédiée aux problèmes de job-shop et flow-shop qui est apparue comme meilleure lors d'expérimentations préliminaires.

Le tableau 7.3 récapitule les résultats sur 4 jeux de 10 instances pour le problème de job-shop proposés par Taillard [103] Les jeux correspondent à des tailles de problème ($n \times m$) : 20×15, 20×20, 30×15 et 30×20. Le nombre d'exécutions pour chaque instance et chaque algorithme est réduit à 10, car les instances sont rarement résolues optimalement dans la limite de temps de 3600 secondes (cette expérience a nécessité plus d'un mois de temps CPU). Les résultats sont rapportés sous la forme de valeurs moyennes sur chaque jeu d'instances. Concernant la valeur de l'objectif (*Makespan*), nous considérons pour chaque instance sa valeur moyenne (Avg), sa meilleure valeur trouvée (Best) et son écart-type (Std Dev). En suivant le protocole de [123], nous comparons aussi les deux approches en terme d'erreurs relatives moyennes (*mean relative error* – MRE) par rapport à la meilleure borne supérieure connue pour ces instances. MRE est la moyenne arithmétique de l'erreur relative de toutes les exécutions sur toutes les instances :

$$MRE(a, K, R) = \frac{1}{|R||K|} \sum_{r \in R} \sum_{k \in K} \frac{c(a, k, r) - c^*(k)}{c^*(k)}$$

où K est l'ensemble des instances, R est l'ensemble des exécutions, $c(a, k, r)$ est la valeur de l'objectif trouvée par l'algorithme a sur l'instance k pendant l'exécution r, et $c^*(k)$ est la meilleure borne supérieure connue pour l'instance k.

	`Ilog Scheduler`					`mistral`				
	Makespan			MRE		Makespan			MRE	
Name	Avg	Best	Std Dev	Avg	Best	Avg	Best	Std Dev	Avg	Best
`tai11-20`	1411.1	1409.9	11.12	0.0336	0.0294	**1409.9**	**1398.1**	23.17	**0.0328**	**0.0241**
`tai21-30`	**1666.0**	1659.0	13.51	**0.0297**	0.0254	1669.9	**1655.4**	25.05	0.0322	**0.0232**
`tai31-40`	1936.1	1927.1	18.67	0.0817	0.0767	**1922.5**	**1898.8**	37.56	**0.0742**	**0.0608**
`tai41-50`	2163.1	2153.1	17.85	0.1055	0.1004	**2143.9**	**2117.6**	44.85	**0.0958**	**0.0824**

TABLE 7.3 – Résultats sur le problème de job-shop.

Comme attendu, les résultats obtenus avec `Ilog Scheduler` sont comparables ou meilleurs que ceux rapportés par Beck [123] puisque nous utilisons une configuration matérielle plus puissante. Les deux approches ont des performances comparables sur les deux premiers jeux d'instance même si les meilleures solutions obtenues par `mistral` sont systématiquement meilleures que celles obtenues avec `Ilog Scheduler`. `mistral` passe mieux à l'échelle sur les deux derniers jeux d'instances composés d'instances de taille supérieure puisque les moyennes des objectifs moyens sont meilleures que les moyennes des meilleurs objectifs obtenues avec `Ilog Scheduler`. Par contre, le modèle *Light* est moins robuste puisque l'écart-type de `mistral` est plus important que celui d'`Ilog Scheduler`. En contrepartie, on peut espérer que le calcul parallèle sera plus bénéfique à `mistral` qu'à `Ilog Scheduler`.

7.4 Conclusion

Dans ce chapitre, nous avons montré que contrairement à une croyance répandue, les problèmes d'atelier peuvent être efficacement résolus par la combinaison d'un modèle offrant une propagation naïve et une heuristique simple inspirée du degré pondéré par l'algorithme décrit au chapitre 6. Nous avons

montré que cette approche surpassait les résultats de l'état de l'art sur de nombreuses instances d'open-shop et de job-shop. Par contre, ces performances restent inférieures à celles de l'algorithme *solution guided multi point constructive search* (SGMPCS) sur les problèmes de job-shop [123].

À la suite de ce travail, Grimes et Hebrard [170] ont montré la pertinence de cette approche sur deux variantes du problème de job-shop avec temps d'attente minimaux ou maximaux. Un mécanisme de guidage similaire à celui SGPMCS a aussi été incorporé. Cependant, une analyse détaillée de l'apprentissage des poids a montré que l'apprentissage des poids pouvait avoir un impact négatif sur une des deux variantes.

Ces modèles peuvent d'ores et déjà être reproduits dans `choco` (voir chapitre 9). Dans de futurs travaux, nous souhaitons comparer les modèles *Light* et *Heavy* sur ces deux variantes mais aussi sur d'autres avec des contraintes de fenêtre de temps qui sont a priori plus favorables aux inférences fortes du modèle *Heavy*.

Chapitre 8

Minimisation du retard algébrique maximal sur une machine à traitement par fournées

Ce chapitre présente une approche en programmation par contraintes pour le problème de minimisation du retard algébrique maximal sur une machine à traitement par fournées (voir chapitre 5). Le modèle proposé exploite une décomposition du problème : rechercher un placement réalisable des tâches dans les fournées ; minimiser le retard algébrique maximal des fournées sur une machine. Ce modèle est enrichi par une nouvelle contrainte globale pour l'optimisation qui utilise une relaxation du problème pour appliquer des techniques de filtrage basé sur les coûts. Les résultats expérimentaux démontrent l'efficacité de ces dernières. Des comparaisons avec d'autres approches exactes montrent clairement l'intérêt de notre approche : notre contrainte globale permet de résoudre des instances dont la taille est supérieur d'un ordre de grandeur par rapport à une formulation en programmation mathématique ou à un branch-and-price.

Sommaire

8.1 Modèle en programmation par contraintes . . . 78
8.2 Description de la contrainte globale `sequenceEDD` . . . 79
8.2.1 Relaxation du problème . . . 80
8.2.2 Filtrage de la variable objectif . . . 81
8.2.3 Filtrage basé sur le coût du placement d'une tâche . . . 82
8.2.4 Filtrage basé sur le coût du nombre de fournées non vides . . . 83
8.3 Stratégie de branchement . . . 84
8.4 Expérimentations . . . 85
8.4.1 Performance des règles de filtrage . . . 85
8.4.2 Performance des heuristiques de sélection de valeur . . . 87
8.4.3 Comparaison avec un modèle en programmation mathématique . . . 88
8.4.4 Comparaison avec une approche par branch-and-price . . . 89
8.5 Conclusion . . . 89
Annexe 8.A : un algorithme quadratique pour le filtrage du placement d'une tâche . . . 90

DANS ce chapitre, nous présentons nos travaux [8, 9] portant sur la minimisation du retard algébrique maximal d'un nombre fini de tâches de tailles différentes réalisées sur une machine à traitement par fournées. Une machine à traitement par fournées peut exécuter simultanément plusieurs tâches regroupées dans une fournée. Ces machines constituent souvent des goulots d'étranglement pour la planification de la production à cause des temps d'exécution élevés des tâches sur ce type de machine.
Nous nous intéressons dans ce chapitre plus particulièrement au problème $1|p\text{-}batch; b < n; non\text{-}identical|L_{max}$ introduit au chapitre 5. On considère un ensemble J de n tâches et une machine à traitement par fournées de capacité b. Chaque tâche j est caractérisée par un triplet d'entiers positifs (p_j, d_j, s_j), où p_j est sa durée

d'exécution, d_j est sa date échue, et s_j est sa taille. La machine à traitement par fournées peut traiter plusieurs tâches simultanément tant que la somme de leurs tailles ne dépasse pas sa capacité b. Toutes les données sont déterministes et connues a priori. La date de fin C_j d'une tâche est la date de fin de la fournée à laquelle elle appartient. Le critère d'optimalité étudié est la minimisation du retard algébrique maximal : $L_{max} = \max_{1 \leq j \leq n}(C_j - d_j)$. Ce problème est NP-difficile *au sens fort* puisque Brucker *et al.* [124] ont prouvé que le même problème avec des tâches de tailles identiques était NP-difficile *au sens fort*.
Ce chapitre est organisé de la manière suivante. La section 8.1 introduit notre modèle en programmation par contraintes. La section 8.2 décrit une nouvelle contrainte globale pour l'optimisation basée sur la résolution d'une relaxation du problème original qui intègre des techniques de filtrage basé sur les coûts. L'algorithme de recherche inspiré par des stratégies classiques en placement à une dimension utilisant une nouvelle heuristique de sélection de valeur est décrit en section 8.3. Finalement, la section 8.4 évalue les performances des composantes de notre approche avant de la comparer à une formulation en programmation mathématique et à une approche par *branch-and-price*.
Nous supposons dans le reste de ce chapitre que les tâches sont triées par taille décroissante ($s_j \geq s_{j+1}$) et que la taille de chaque tâche est inférieure à la capacité de la machine ($s_j \leq b$). On peut noter de plus que le nombre de fournées m est inférieur au nombre de tâches n.

8.1 Modèle en programmation par contraintes

Notre modèle est basé sur une décomposition du problème en un problème de placement des tâches dans les fournées et un problème de minimisation du retard algébrique maximal des fournées sur une machine. Le problème du placement des tâches dans les fournées est équivalent à un problème de placement à une dimension, connu pour être NP-complet (*bin packing*, voir section 3.5). L'énoncé de ce problème est le suivant : soit n articles (tâches) caractérisés chacun par une taille positive $s_j \geq 0$ et m conteneurs (fournées) caractérisés par une capacité b, existe-t'il un placement des n articles dans les m conteneurs tel que la somme des tailles des articles dans chaque conteneur reste inférieure à la capacité b ? Une fois que les tâches ont été placées dans les fournées, l'ordonnancement des fournées doit minimiser le retard algébrique maximal des fournées sur une machine. Ce problème, noté $1||L_{max}$, peut être résolu en temps polynomial [153] : un ordonnancement optimal est obtenu en appliquant la règle de Jackson, aussi appelée règle EDD (*earliest due date (EDD-)rule*) qui séquence les tâches en ordre non décroissant de leur date échue [154].

Nous décrivons maintenant le modèle PPC du problème étudié. Soit $J = [1, n]$ l'ensemble des indices des tâches et $K = [1, m]$ l'ensemble des indices des fournées. La valeur maximale des dates échues est notée $d_{max} = \max_J\{d_j\}$ et celle des durées d'exécution est notée $p_{max} = \max_J\{p_j\}$. La variable $B_j \in K$ représente l'indice de la fournée dans laquelle est placée la tâche j, et la variable $J_k \subseteq J$ représente l'ensemble des tâches contenues dans la fournée k. Ces deux ensembles de variables satisfont les contraintes de liaison : $\forall j \in J,\ B_j = k \Leftrightarrow j \in J_k$. Les variables entières positives $P_k \in [0, p_{max}]$, $D_k \in [0, d_{max}]$ et $S_k \in [0, b]$ représentent respectivement la durée d'exécution, la date échue, et la charge de la fournée k. Finalement, les variables $M \in [0, m]$ et L_{max} (non bornée) représentent respectivement le nombre de fournées non vides et l'objectif. Le modèle en programmation par contraintes est le suivant :

$$\texttt{maxOfASet}\left(P_k,\ J_k,\ [p_j]_J,\ 0\right) \qquad \forall k \in K \tag{8.1}$$

$$\texttt{minOfASet}\left(D_k,\ J_k,\ [d_j]_J,\ d_{max}\right) \qquad \forall k \in K \tag{8.2}$$

$$\texttt{pack}\left([J_k]_K,\ [B_j]_J,\ [S_k]_K,\ M,\ [s_j]_J\right) \tag{8.3}$$

$$\texttt{sequenceEDD}\left([B_j]_J,\ [D_k]_K,\ [P_k]_K,\ M,\ L_{max}\right) \tag{8.4}$$

Les contraintes (8.1), où $[p_j]_J$ est le tableau contenant les durées d'exécution, impose que la durée P_k d'une fournée k est la plus grande durée d'exécution de ses tâches (ensemble J_k) si la fournée n'est pas vide, et égale à 0 sinon. De la même manière, les contraintes (8.2) imposent que la date échue D_k de la fournée k est la plus petite date échue minimale de ses tâches, et égale à d_{max} sinon. En effet, le retard algébrique d'une fournée k défini par la relation $\max_{j \in J_k}(C_k - d_j)$ est égal à $C_k - \min_{j \in J_k}(d_j)$ où C_k est la date d'achèvement de la fournée k. Remarquez que le retard algébrique d'une tâche peut être négatif.

La contrainte (8.3) est une variante de la contrainte globale proposée par Shaw [70] pour le problème de *bin packing*. La contrainte applique des règles de filtrage inspirées de raisonnements sur les problèmes de sac à dos ainsi qu'une borne inférieure dynamique sur le nombre minimal de conteneurs non vides. Elle remplace les contraintes de liaison entre les variables représentant le placement des tâches dans les fournées ($\forall j \in J,\ B_j = k \Leftrightarrow j \in J_k$) et assure la cohérence entre le placement des tâches et les charges des conteneurs ($\forall k \in K,\ \sum_{j \in J_k} s_j = S_k$). De plus, `pack` réduit les domaines en tenant compte d'une contrainte redondante assurant la cohérence entre les charges des conteneurs et les tailles des tâches ($\sum_J s_j = \sum_K S_k$). Remarquez que le respect de la capacité de la machine est assuré par le domaine initial des variables S_k. Notre implémentation de la contrainte `pack` est décrite dans la section 9.3.9.
La contrainte globale (8.4) impose que la valeur de la variable objectif L_{max} soit égale au retard algébrique maximal des fournées lorsqu'elles sont ordonnancées en suivant la règle EDD. Cette contrainte applique plusieurs règles de filtrages détaillées dans la section 8.2.
Pour conclure, une solution de notre modèle PPC est composée d'un placement réalisable des tâches dans les fournées, et du retard algébrique maximal associé à l'instance du problème $1||L_{max}$ engendrée par les fournées. Remarquez que ce modèle reste valide en présence de contraintes additionnelles imposant des capacités non identiques des conteneurs, des incompatibilités entre les tâches, ou une répartition équilibrée de la charge des conteneurs. En effet, il suffit de modifier les domaines des variables associées pour prendre en compte ces nouvelles contraintes.

Korf [69] a observé que la prise en compte de conditions de dominance entre les placements réalisables, qui permettent de n'en considérer qu'un nombre restreint, est un élément crucial pour la résolution des problèmes de placement à une dimension. Les méthodes efficaces pour ces problèmes (voir section 3.5) utilisent intensivement des conditions d'équivalence et de dominance pour réduire l'espace de recherche. Mais, ces procédures considèrent généralement que les articles avec des tailles identiques ou les conteneurs avec des charges identiques peuvent être permutés sans dégrader la qualité de la solution. Dans le contexte d'une machine à traitement par fournées, une permutation entre deux tâches de tailles identiques peut modifier la qualité de la solution, car cette permutation peut entraîner une modification des durées d'exécution ou des dates échues des fournées. Par conséquent, ces règles ne peuvent pas être appliquées dans ce contexte. De la même manière, les méthodes par *bin completion* [66, 69, 71] ne peuvent pas être appliquées, car elles utilisent des conditions de dominance en considérant uniquement la charge des conteneurs.
Par contre, on peut toujours réduire l'espace de recherche en imposant que deux tâches i et j telles que $s_i + s_j > b$ appartiennent à des fournées différentes. Puisque les tâches sont triées par taille décroissante, notons j_0 le plus grand indice tel que chaque paire de tâches dont les indices sont inférieurs à j_0 appartiennent obligatoirement à des fournées différentes : $j_0 = \max\{j \mid \forall\ 1 \leq i < j,\ s_{i+1} + s_i > b\}$. Les contraintes (8.5), rangeant consécutivement les plus grandes tâches dans les premiers conteneurs, peuvent être ajoutées au modèle.

$$B_j = j \qquad 1 \leq j \leq j_0 \tag{8.5}$$

8.2 Description de la contrainte globale `sequenceEDD`

Les réductions de domaine proviennent généralement de raisonnements liés à la satisfiabilité. Dans le contexte de l'optimisation, les réductions de domaine peuvent aussi être réalisées en se basant sur des raisonnements liés à l'optimalité. Le filtrage tend alors à retirer des combinaisons de valeurs qui ne peuvent pas être impliquées dans une solution améliorant la meilleure solution découverte jusqu'à présent. Focacci *et al.* [155] ont proposé d'inclure ces raisonnements dans une contrainte globale qui représente une relaxation d'un sous-problème ou d'elle-même. Des composants internes à la contrainte globale fournissent des algorithmes de recherche opérationnelle calculant la solution optimale d'une relaxation du problème ainsi qu'une fonction estimant le gradient (variation) du coût de cette solution optimale après l'affectation d'une valeur à une variable. Focacci *et al.* ont montré l'intérêt d'utiliser cette information pour le filtrage, mais aussi pour guider la recherche sur plusieurs problèmes d'optimisation combinatoire.

Comme mentionné précédemment, la contrainte globale `sequenceEDD` assure que la valeur de la variable L_{max} soit le plus grand retard algébrique de la séquence de fournées ordonnancées en suivant la règle EDD. Cette contrainte utilise une relaxation du problème original fournissant une borne inférieure sur la valeur de la variable objectif pour réduire l'espace de recherche. L'idée principale est de déduire des

contraintes primitives à partir des informations sur les coûts des affectations. Un premier composant de la contrainte globale fournit donc une solution optimale pour une relaxation du problème qui consiste à minimiser le retard algébrique maximal des fournées sur une machine. Un autre composant est une fonction gradient donnant une estimation (borne inférieure) du coût à ajouter à celui de la solution optimale lors de l'affectation d'une valeur à certaines variables. Ainsi, la solution optimale de la relaxation permet d'améliorer la borne inférieure de la variable objectif, puis d'éliminer des parties de l'espace de recherche pour lesquelles les bornes inférieures sont supérieures à la meilleure solution découverte jusqu'à présent. La relaxation du problème est présentée dans la section 8.2.1 suivie de quatre règles de filtrage présentées dans les sections 8.2.2, 8.2.3 and 8.2.4.

8.2.1 Relaxation du problème

Dans cette section, nous décrivons la construction d'une instance $I(\mathcal{A})$ de la relaxation, c'est-à-dire une instance du problème $1||L_{max}$, à partir d'une affectation partielle $\mathcal{A}$ des variables, puis un algorithme pour la résoudre. Notons $\delta_1, \delta_2, \ldots, \delta_l$ les valeurs distinctes des dates échues d_j des jobs $j \in J$ triées en ordre croissant. Le nombre l de dates échues distinctes peut être inférieur au nombre n de tâches lorsque certaines dates échues sont identiques. Par souci de simplicité, nous supposerons dans les formules et algorithmes de ce chapitre que $l = n$. En section 2.1 page 10, nous avons défini une affectation partielle $\mathcal{A}$ comme l'ensemble des domaines courants de toutes les variables ainsi que les notations propres aux réductions de domaine, par exemple une instanciation $x \leftarrow v$. Soit $K_{\mathcal{R}p}(\mathcal{A}) = \{k \in K \mid \max(D_k)\ \mathcal{R}\ \delta_p\}$ l'ensemble des fournées liées à la date échue δ_p par la relation arithmétique $\mathcal{R} \in \{<, \leq, =, \geq, >\}$. De la même manière, $J_{\mathcal{R}p}(\mathcal{A}) = \{j \in J \mid d_j\ \mathcal{R}\ \delta_p\}$ est l'ensemble des tâches liées à la date échue δ_p par la relation $\mathcal{R}$. Finalement, notons $P(\mathcal{A}, \tilde{K}) = \sum_{k \in \tilde{K}} \min(P_k)$ la durée totale minimale d'un ensemble de fournées $\tilde{K} \subseteq K$ pour l'affectation partielle $\mathcal{A}$.
Une instance $I(\mathcal{A})$ de la relaxation est composée de n blocs où chaque bloc $p \in J$ contient l'ensemble de fournées $K_{=p}(\mathcal{A})$. Chaque bloc p a une date échue δ_p et une durée d'exécution $\pi_p(\mathcal{A}) = P(\mathcal{A}, K_{=p}(\mathcal{A}))$. Étant donné que les blocs sont, par définition, numérotés en ordre croissant ($\delta_p < \delta_{p+1}$), le retard algébrique maximal de $I(\mathcal{A})$ est celui de la séquence de blocs ordonnancés dans cet ordre. Notons $C_p(\mathcal{A}) = \sum_{q=1}^{p} \pi_q(\mathcal{A})$ et $L_p(\mathcal{A}) = C_p(\mathcal{A}) - \delta_p$ la date d'achèvement et le retard algébrique du bloc p dans cette séquence. Le retard algébrique maximal d'une solution optimale de l'instance $I(\mathcal{A})$ est alors le suivant : $L(\mathcal{A}) = \max_J(L_p(\mathcal{A}))$.

La figure 8.1 illustre la construction et la résolution de la relaxation du problème $1|p\text{-}batch; b < n; non\text{-}identical|L_{max}$ à deux étapes différentes de la résolution. La capacité de la machine à traitement par fournées est $b = 10$. Le tableau 8.1(a) décrit les tâches à ordonnancer dans le problème original. Le tableau 8.1(b) donne les blocs de l'instance $I(\mathcal{A}_1)$ où $\mathcal{A}_1$ est une solution du problème, c'est-à-dire une affectation totale. Dans cet exemple, il y a moins de blocs ($l = 3$) que de tâches ($n = 4$). La figure 8.1(d) représente la solution optimale de la relaxation $I(\mathcal{A}_1)$ qui est également une solution optimale du problème original. La machine à traitement par fournées est représentée par un diagramme dans lequel les axes horizontaux et verticaux correspondent respectivement au temps et à la charge de la ressource. Une tâche est dessinée comme un rectangle dont la longueur et la hauteur représentent respectivement sa durée et sa taille. Les tâches dont les dates de début sont identiques appartiennent à la même fournée. La solution contient donc trois fournées : la première contient uniquement la tâche 1 ; la seconde contient les tâches 2 et 4 (sa charge est égale à $s_2 + s_4 = 9$, sa durée minimale est égale à $\max\{p_2, p_4\} = 9$ et sa date échue maximale est égale à $\min\{d_2, d_4\} = 2$) ; la troisième contient uniquement la tâche 3. Un bloc est représenté comme un rectangle dessiné en pointillés entourant les fournées lui appartenant. Le retard algébrique d'un bloc est représenté sur la droite de la figure par une ligne allant de sa date échue à sa date d'achèvement. Dans cet exemple, le bloc 1 contient les fournées 1 et 2 puisque $\max(D_1) = \max(D_2) = d_1$. Le bloc 2 est vide, car qu'il n'y a aucune fournée k telle que $\max(D_k) = \delta_2$. Le retard algébrique d'un bloc vide, dessiné comme une ligne en pointillés, est dominé par celui de son premier prédécesseur non vide. Le bloc 3 contient la fournée 3 constituée uniquement de la tâche 3.
Le tableau 8.1(c) donne les blocs de l'instance $I(\mathcal{A}_2)$ où $\mathcal{A}_2$ est une affectation partielle (la tâche 4 n'est pas encore placée). La figure 8.1(e) représente la solution optimale de l'instance $I(\mathcal{A}_2)$ dans laquelle la tâche 4 n'apparaît donc pas. Dans cet exemple, chaque bloc contient une seule fournée : le premier, le deuxième et le troisième bloc contiennent respectivement les fournées 1, 2 et 3. En fait, $\mathcal{A}_2$ est déduite

Job	1	2	3	4
s_j	8	7	5	2
p_j	5	8	7	9
d_j	2	7	10	2

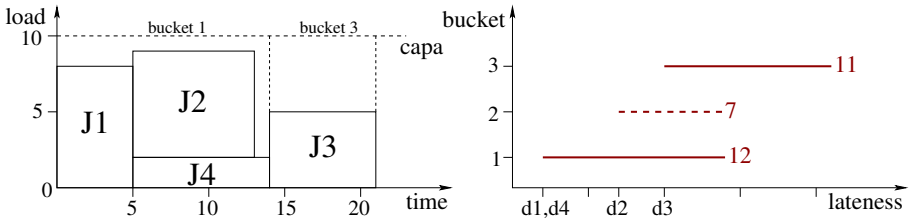
(a) Instance de $1||L_{max}$.

Bloc	1	2	3
δ_p	2	7	10
$\pi_p(\mathcal{A})$	14	0	7

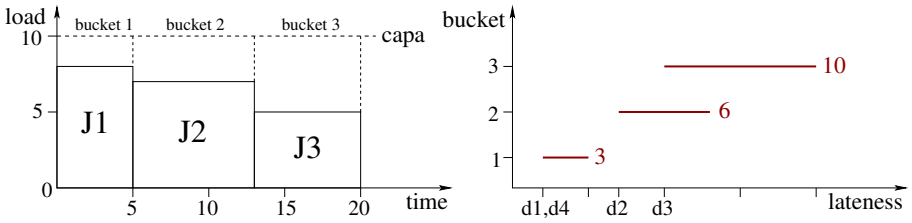
(b) Relaxation $I(\mathcal{A}_1)$.

Bloc	1	2	3
δ_p	2	7	10
$\pi_p(\mathcal{A})$	5	8	7

(c) Relaxation $I(\mathcal{A}_2)$.



(d) Solution de la relaxation associée à la solution (affectation totale) $\mathcal{A}_1$. $\mathcal{A}_1 = \{B_1 \leftarrow 1, B_2 \leftarrow 2, B_3 \leftarrow 3, B_4 \leftarrow 2\}$



(e) Solution de la relaxation associée à l'affectation partielle $\mathcal{A}_2$. La tâche 4 est absente, car elle n'est pas encore placée. $\mathcal{A}_2 = \{B_1 \leftarrow 1, B_2 \leftarrow 2, B_3 \leftarrow 3, B_4 \in [1, 4]\}$.

FIGURE 8.1 – Deux exemples de construction de la relaxation $I(\mathcal{A})$.

pendant la propagation à la racine de l'arbre de recherche par les contraintes (8.5) qui imposent que $B_1 = 1$, $B_2 = 2$, and $B_3 = 3$.

À chaque étape de la résolution, la construction de l'instance $I(\mathcal{A})$ peut être réalisée en $O(n)$. En effet, les dates échues sont triées en ordre croissant une fois pour toutes avant le début de la résolution avec une complexité en $O(n \log n)$. Ensuite, la construction de l'instance $I(\mathcal{A})$ consiste à affecter chaque fournée k à un bloc p. Ceci est réalisable en temps constant puisque le bloc p d'une fournée k est le bloc p vérifiant la condition $\delta_p = \max(D_k)$. L'insertion d'une fournée k dans son bloc et la mise à jour de la durée d'exécution du bloc sont aussi réalisées en temps constant. Par conséquent, la complexité totale de la construction de l'instance est $O(n)$, car le nombre de fournées m est inférieur au nombre de tâches n. Finalement, la résolution de l'instance $I(\mathcal{A})$ consiste simplement à séquencer les blocs en suivant leur numérotation, ce qui peut être réalisé en $O(n)$. Les sections suivantes présentent les règles de filtrages basées sur la relaxation du problème.

8.2.2 Filtrage de la variable objectif

Les règles de filtrage portant sur la variable objectif L_{max} reposent sur les deux propositions suivantes.

Proposition 1. *L est une fonction monotone d'un ensemble partiellement ordonné d'affectations vers les entiers naturels :*

$$\mathcal{A}' \subseteq \mathcal{A} \Rightarrow L(\mathcal{A}') \geq L(\mathcal{A}).$$

Démonstration. Les contraintes (8.1) imposent que la durée minimale d'une fournée dans $\mathcal{A}'$ soit supérieure à sa durée minimale dans $\mathcal{A}$, la durée totale d'un ensemble de fournées $\tilde{K}$ ne peut pas décroître lors de la restriction de $\mathcal{A}$ à $\mathcal{A}'$: $\forall \tilde{K} \subseteq K,\ P(\mathcal{A}, \tilde{K}) \leq P(\mathcal{A}', \tilde{K})$. De plus, puisque les contraintes (8.2) imposent que le placement d'une nouvelle tâche dans une fournée appartenant au bloc p implique son transfert vers un bloc p' tel que $p' \leq p$, l'ensemble des fournées appartenant au bloc p ou à un de ces

prédécesseurs ne peut pas diminuer lors de la restriction de $\mathcal{A}$ à $\mathcal{A}'$: $\forall p \in J,\ K_{\leq p}(\mathcal{A}) \subseteq K_{\leq p}(\mathcal{A}')$. Parallèlement, puisque P_k est une variable entière positive, $P(\mathcal{A}, \tilde{K})$ est une fonction croissante des sous-ensembles de fournées vers les entiers positifs : $\tilde{K} \subseteq \tilde{K}' \Rightarrow P(\mathcal{A}, \tilde{K}) \leq P(\mathcal{A}, \tilde{K}')$. Par conséquent, les inégalités suivantes sont valides :

$$\begin{aligned} C_p(\mathcal{A}) &= \sum_{1 \leq q \leq p} \pi_q(\mathcal{A}) = P(\mathcal{A}, K_{\leq p}(\mathcal{A})) \\ &\leq P(\mathcal{A}', K_{\leq p}(\mathcal{A})) \leq P(\mathcal{A}', K_{\leq p}(\mathcal{A}')) \\ &\leq C_p(\mathcal{A}'). \end{aligned}$$

Étant donné que la date d'achèvement d'un bloc p ne peut qu'augmenter lors de la restriction de $\mathcal{A}$ à $\mathcal{A}'$, la monotonie de la fonction $I(\mathcal{A})$ est prouvée de la manière suivante :

$$(8.1) \wedge (8.2) \quad \Rightarrow \quad \forall p \in J,\ C_p(\mathcal{A}') \geq C_p(\mathcal{A}) \quad \Rightarrow \quad L(\mathcal{A}) \leq L(\mathcal{A}')$$

□

Proposition 2. *Lorsque toutes les durées d'exécution et toutes les dates échues des fournées sont fixées, un ordonnancement optimal de la relaxation $I(\mathcal{A})$ correspond aussi à un ordonnancement optimal pour toute solution réalisable obtenue à partir de $\mathcal{A}$.*

Démonstration. Lorsque toutes les durées d'exécution et dates échues des fournées sont fixées, les durées et retards des blocs ne seront plus modifiés jusqu'à ce qu'une solution réalisable soit découverte. Par conséquent, la proposition est vraie si le retard algébrique maximal de la relaxation $L(\mathcal{A})$ est égal au retard algébrique maximal de toute solution obtenue à partir de $\mathcal{A}$. Comme toutes les fournées d'un bloc p ont la même date échue δ_p, toutes les séquences de ses fournées sont équivalentes pour la règle EDD. Ainsi, un ordonnancement optimal des blocs dans la relaxation correspond à une classe d'ordonnancement optimale des fournées (respectant la règle EDD) pour le problème initial. Remarquez qu'il est souvent préférable d'ordonnancer les fournées d'un bloc par ordre croissant de durée d'exécution puisque cela diminue le retard moyen des fournées. □

Grâce à la proposition 2, nous déduisons la règle de filtrage du retard final (*final lateness filtering rule* – FF) qui résout la relaxation pour instancier la variable objectif dès que toutes les durées et toutes les dates échues des fournées sont instanciées :

$$\forall k \in K, P_k \text{ et } D_k \text{ sont instanciées } \Rightarrow L_{max} \leftarrow L(\mathcal{A}). \tag{FF}$$

Remarquez que cette règle peut se déclencher avant le placement de toutes les tâches, car la propagation peut réduire les domaines à un singleton. Un corollaire des propositions 1 et 2 est qu'à chaque étape de la résolution, le retard algébrique maximal $L(\mathcal{A})$ de l'instance de la relaxation $I(\mathcal{A})$ est une borne inférieure du retard algébrique maximal de tout ordonnancement restreignant $\mathcal{A}$. Par conséquent, la valeur minimale de l'objectif peut être mise à jour après chaque modification pertinente des domaines par la règle de filtrage du retard (*lateness filtering rule* – LF) :

$$\exists k \in K, \min(P_k) \text{ ou } \max(D_k) \text{ ont changé} \Rightarrow \min(L_{max}) \leftarrow L(\mathcal{A}) \tag{LF}$$

8.2.3 Filtrage basé sur le coût du placement d'une tâche

La règle de filtrage basée sur le coût du placement d'une tâche (*cost-based domain filtering rule of assignments* – AF) réduit l'espace de recherche en se basant sur le coût marginal associé au placement d'une tâche j dans une fournée k ($B_j \leftarrow k$). L'idée est d'éliminer, après chaque modification pertinente des domaines, chaque tâche j comme candidate pour un placement dans la fournée k, si le coût marginal associé à son placement dans la fournée k dépasse la meilleure borne supérieure découverte jusqu'à présent, ou plus formellement :

$$\begin{aligned} &\exists k \in K, \min(P_k) \text{ ou } \max(D_k) \text{ ont changé} \Rightarrow \\ &\qquad \forall j \in J, \text{ tel que } |\mathcal{D}(B_j)| > 1 \text{ et } \forall k \in \mathcal{D}(B_j), \\ &\qquad\qquad L(\mathcal{A} \cap \{B_j \leftarrow k\}) > \max(L_{max}) \quad \Rightarrow \quad B_j \not\leftarrow k. \end{aligned} \tag{AF}$$

$\mathcal{A} \cap \{B_j \leftarrow k\}$ est une abréviation pour $\mathcal{A} \cap \{B_j \leftarrow k, \min(P_k) \leftarrow p_j, \max(D_k) \leftarrow d_j\}$. En effet, la propagation des contraintes (8.1) et (8.2) après le placement $B_j \leftarrow k$ implique que $\min(P_k) \leftarrow p_j$ et $\max(D_k) \leftarrow d_j$.
Un algorithme de filtrage basique peut calculer le coût marginal de chaque placement réalisable avec une complexité $O(n^3)$. Dans la section 8.5, nous proposons une version de cet algorithme de complexité $O(nm)$ basée sur un calcul incrémental des coûts marginaux.

8.2.4 Filtrage basé sur le coût du nombre de fournées non vides

Dans cette section, nous introduisons une règle de filtrage basée sur le coût marginal associé au nombre de fournées non vides M. Notons $K^\star = \{k \in K \mid \exists j \in J,\ B_j = k\}$ l'ensemble des fournées non vides. Nous supposons que $|K^\star| = \max\{k \in K^\star\}$, c'est-à-dire les tâches sont placées dans les premières fournées. Notons $J^\star = \{j \in J \mid \mathcal{D}(B_j) > 1 \wedge \max(B_j) > |K^\star|\}$ l'ensemble des tâches *disponibles*, c'est-à-dire les tâches qui peuvent être placées dans une nouvelle fournée. La règle (PF1) actualise le nombre possible de fournées non vides en considérant leur nombre actuel et le nombre de tâches disponibles.

$$\min(M) \leftarrow |K^\star| \qquad \max(M) \leftarrow |K^\star| + |J^\star| \tag{PF1}$$

Cette règle (appliquée aussi par `pack`) est nécessaire pour garantir la validité des raisonnements présentés ci-dessous. Soit $\mathcal{A}' = \mathcal{A} \cap \left(\cup_{j \in \tilde{J}} \{B_j \leftarrow k_j\}\right)$ une affectation partielle restreinte par le placement d'un sous-ensemble $\tilde{J} \subseteq J^\star$ de tâches disponibles à de nouvelles fournées, c'est-à-dire un ensemble de fournées vides avec des indices distincts ($\forall j \in \tilde{J},\ k_j > |K^\star|$ et $\forall i \neq j \in \tilde{J},\ k_i \neq k_j$). Par conséquent, la date d'achèvement et le retard algébrique de chaque bloc p doivent être actualisés en fonction de l'augmentation de la durée de ces prédécesseurs. En effet, la nouvelle date d'achèvement d'un bloc p est égale à sa date d'achèvement avant les placements à laquelle s'ajoute l'augmentation de sa durée et celles de ses prédécesseurs : $C_p(\mathcal{A}') = C_p(\mathcal{A}) + \sum_{j \in \tilde{J}_{\leq p}} p_j$. Malheureusement, la combinatoire des placements possibles augmente exponentiellement ce qui rend ce type de filtrage difficile et coûteux.
Pour pallier cet inconvénient, nous calculons une borne inférieure de $C_p(\mathcal{A}')$ en calculant une borne inférieure sur la date d'achèvement $C_p(\mathcal{A} \cap \{M \leftarrow q\})$ d'un bloc p pour un nombre total q de fournées non vides de la manière suivante. Notons $\Pi(q)$ la somme des q plus petites durées parmi celles des tâches disponibles. Si on crée $q - |K^\star|$ nouvelles fournées avec des tâches disponibles, il y a au moins $q - |K^\star| - |J^\star_{>p}|$ nouvelles fournées ordonnancées dans ou avant le bloc p. Dans ce cas, la date d'achèvement d'un bloc p après la création de $q - |K^\star|$ fournées est supérieure à sa date d'achèvement avant la création des nouvelles fournées additionnée à la somme minimale $\Pi(q - |K^\star| - |J^\star_{>p}|)$ des durées d'exécution de $q - |K^\star| - |J^\star_{>p}|$ tâches disponibles :

$$\begin{aligned} C_p(\mathcal{A} \cap \{M \leftarrow q\}) &= C_p(\mathcal{A}) + \Pi(q - |K^\star| - |J^\star_{>p}|) \\ &\leq C_p\left(\mathcal{A} \cap \left(\cup_{j \in \tilde{J}} \{B_j \leftarrow k_j\}\right)\right) \qquad \forall \tilde{J} \subseteq J^\star,\ |\tilde{J}| = q - |K^\star|. \end{aligned}$$

Ainsi, le retard algébrique maximal d'une solution obtenue à partir de $\mathcal{A}$ contenant exactement q fournées non vides est supérieur à $L(\mathcal{A} \cap \{M \leftarrow q\})$. La règle (PF2) actualise la borne inférieure de l'objectif en fonction du coût marginal associé au nombre minimal de fournées non vides. La règle (PF3) décrémente le nombre maximal de fournées non vides tant que son coût marginal dépasse la meilleure borne supérieure découverte jusqu'à présent.

$$\min(L_{max}) \leftarrow L(\mathcal{A} \cap \{M \leftarrow \min(M)\}) \tag{PF2}$$
$$L(\mathcal{A} \cap \{M \leftarrow \max(M)\}) > \max(L_{max}) \Rightarrow \max(M) \leftarrow \max(M) - 1 \tag{PF3}$$

Après chaque modification pertinente des domaines, la règle de filtrage basée sur le coût du nombre de fournées non vides (*cost-based domain filtering rule based on bin packing* – PF) applique les trois règles de filtrage décrites ci-dessus.

$$(\exists j \in J, \mathcal{D}(B_j) \text{ a changé}) \vee (\exists k \in K, \min(P_k) \text{ ou } \max(D_k) \text{ ont changé}) \Rightarrow$$
$$\text{Apply rules (PF1), (PF2) and (PF3)} \tag{PF}$$

Le calcul de la fonction Π est réalisé en $O(n \log n)$ pendant l'initialisation en triant et sommant les durées d'exécution des tâches disponibles. Les règles (PF1) et (PF2) sont appliquées en temps linéaire, puisque la construction et la résolution de $I(\mathcal{A})$ a une complexité en $O(n)$. La règle (PF3) est appliquée jusqu'à ce que le coût marginal associé au nombre maximal de fournées non vides devienne satisfiable, c'est-à-dire au plus $|J^\star|$ fois. Par conséquent, la complexité du filtrage basé sur le coût du nombre de fournées non vides est $O(n^2)$.

La figure 8.2 illustre le calcul du coût marginal associé à la présence d'exactement $q = 4$ fournées non vides pour l'instance introduite dans le tableau 8.1(a) et une affectation vide $\mathcal{A}_3$ (avant la propagation au nœud racine). L'ensemble des tâches disponibles $J^\star$ est l'ensemble des tâches J et toutes les fournées sont vides ($K^\star = \emptyset$). Par conséquent, les règles (PF1) et (PF2) ne modifient pas le domaine de M. Puisque la date d'achèvement $C_p(\mathcal{A}_3)$ de chaque bloc p est nul, sa nouvelle date d'achèvement $C_p(\mathcal{A}_3 \cap \{M \leftarrow 4\})$ est égale à $\Pi(q - |J^\star_{>p}|)$. Ainsi, le bloc 1 s'achève à l'instant $\Pi(4-|\{2,3\}|) = p_1+p_3 = 12$, le bloc 2 s'achève à l'instant $\Pi(4-|\{3\}|) = p_1+p_3+p_2 = 20$, et le bloc 3 s'achève à l'instant $\Pi(4-|\emptyset|) = p_1+p_3+p_2+p_4 = 29$. En supposant que la meilleure borne supérieure découverte pendant la résolution soit égale à 15, alors le nombre maximal de fournées non vides est réduit de 4 à 3. On doit alors appliquer à nouveau la règle (PF2) pour $q = 3$.

8.3 Stratégie de branchement

À chaque nœud de l'arbre de recherche, la stratégie de branchement choisit la variable B_j associée à une tâche j en utilisant une heuristique de sélection de variable, et place la tâche j dans la fournée k choisie par une heuristique de sélection de valeur. Lors d'un retour arrière, le branchement interdit le placement de la tâche sélectionnée dans la fournée sélectionnée. Si une fournée redevient vide, une règle d'élimination de symétrie est appliquée pour éliminer les fournées équivalentes, c'est-à-dire les autres fournées vides, comme candidates pour le placement de la tâche courante. Remarquez que cette règle dynamique d'élimination de symétrie généralise l'idée qui est à la base de l'introduction des contraintes (8.5), parce qu'elle évite de placer les j_0 plus grandes tâches dans une autre fournée lors d'un retour arrière.
Plusieurs heuristiques de sélection de variable exploitant les informations sur les durées, tailles et dates échues peuvent être utilisées. Nous avons choisi une heuristique classique appelée *complete decreasing* [67] qui place les tâches par ordre décroissant de taille. En effet, des expérimentations préliminaires ont montré que privilégier le placement des grandes tâches améliore le filtrage des contraintes `pack` et `sequenceEDD` par rapport aux variantes se basant aussi sur les durées d'exécution et les dates échues.
Nous avons sélectionné deux heuristiques de sélection de valeur classiques pour les problèmes de placement à une dimension : *first fit* qui choisit la première fournée possible pour une tâche ; et *best fit* qui choisit la fournée la plus chargée, c'est-à-dire avec le moins d'espace restant. Par ailleurs, nous proposons une nouvelle heuristique de sélection de valeurs appelée *batch fit* qui choisit la fournée ayant la plus petite valeur $\gamma(B_j \leftarrow k)$ où $\gamma(B_j \leftarrow k)$ est une estimation de la compatibilité du placement dans l'instance $I(\mathcal{A})$ calculée de la manière suivante :

$$\gamma(B_j \leftarrow k) = \frac{|\min(P_k) - p_j|}{\max_J(p_j) - \min_J(p_j)} + \frac{|\max(D_k) - d_j|}{\max_J(d_j) - \min_J(d_j)}$$

L'idée sur laquelle repose cette heuristique est qu'une solution « parfaite » est composée de fournées contenant des tâches dont les durées et dates échues sont toutes identiques.

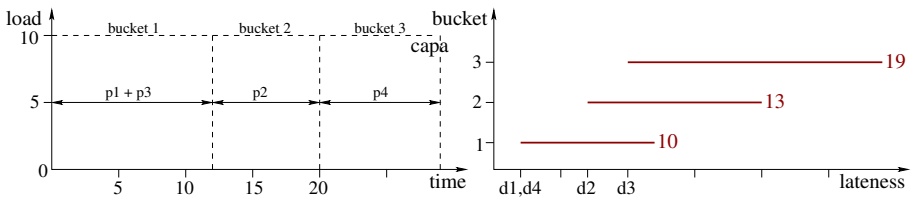


FIGURE 8.2 – Exemple de calcul de $L(\mathcal{A}_3 \cap \{M \leftarrow 4\})$ pour une affectation (vide) $\mathcal{A}_3$.
$\mathcal{A}_3 = \{B_1 \in [1,4], B_2 \in [1,4], B_3 \in [1,4], B_4 \in [1,4]\}$.

8.4 Expérimentations

Cette section récapitule les expérimentations menées pour évaluer notre approche sur le jeu d'instances présenté en section 5.3. Lors de la résolution d'une instance, nous utilisons la procédure d'optimisation top-down décrite en section 2.4, car nous n'avons aucune garantie sur la borne inférieure initiale. Les sections 8.4.1 et 8.4.2 évaluent respectivement les performances des règles de filtrage et heuristiques de sélection de valeur. Finalement, les résultats de notre approche sont comparés à ceux d'une formulation en programmation mathématique en section 8.4.3, puis à une approche par branch-and-price en section 8.4.4. Toutes les expérimentations ont été réalisées sur une grille de calcul composée de machines tournant sous Linux où chaque nœud possède 48 GB de mémoire vive et deux processeurs à quadruple cœur cadencé s à 2.4 GHz. Notre implémentation est basée sur le solveur `choco` (voir chapitre 9) pour lequel la contrainte `pack` est disponible depuis la livraison 2.0.0. Un module externe fournit une implémentation de la contrainte `sequenceEDD` et de notre nouvelle heuristique de sélection de valeur. Ce module utilise l'algorithme pour la règle de filtrage (AF) présentée en section 8.5. Une limite de temps de 3600 secondes (1h) est fixée pour la résolution.

8.4.1 Performance des règles de filtrage

L'amélioration du filtrage d'une contrainte est souvent bénéfique à la résolution, mais il arrive quelquefois que les performances d'un algorithme simple surpassent celles d'algorithmes plus compliqués. Par conséquent, les sections 8.4.1.1 et 8.4.1.2 évaluent respectivement l'efficacité du filtrage et son utilité pendant la résolution (voir section 8.4.1.2). Dans cette section, le nom d'une règle de filtrage est une abréviation qui représente leur structure hiérarchique, c'est-à-dire (FF) $\subset$ (LF) $\subset$ (AF) $\subset$ (PF). Par exemple, l'abréviation (PF) signifie que toutes les règles de filtrage sont appliquées.

8.4.1.1 Calcul d'une borne inférieure destructive

Nous comparons la puissance des règles de filtrage en calculant des bornes inférieures destructives (procédure destructive-lower-bound de la section 2.4). Les trois premières bornes inférieures destructives sont calculées en utilisant les règles de filtrage (LF), (AF), et (PF). Remarquez que nous ignorons délibérément la règle (FF), car il est peu probable qu'elle réduise des domaines dans ce contexte. Ces trois premières bornes étant calculées très rapidement, trois autres (meilleures) bornes sont calculées en intégrant une étape de *shaving* comme suggéré par Rossi *et al.* [10]. Le *shaving* est semblable à une preuve par contradiction. On propage le placement de la tâche j dans la fournée k. Si une insatisfiabilité est détectée, alors le placement n'est pas valide et on peut supprimer la tâche j des candidates pour un placement dans la fournée k. Pour limiter le temps de calcul, *shaving* est utilisé une seule fois pour chaque placement $B_j \leftarrow k$.

Notons ub la meilleure borne supérieure découverte pour une instance donnée [1]. La qualité d'une borne inférieure lb pour une instance donnée est estimée grâce à la formule $(100 \times (lb + d_{max})) \div (ub + d_{max})$ (inspiré des travaux de Malve et Uzsoy [135]). Cette mesure est égale à l'écart à l'optimum pour les instances contenant moins de 20 tâches, car tous les optimums de ces instances ont été prouvés. Par ailleurs, cette mesure est très proche de l'écart à l'optimum pour les instances contenant 50 tâches, car seules deux instances ne sont pas résolues optimalement. Le tableau 8.1 donne la qualité moyenne des deux types de borne inférieure destructive (colonnes 5–7 et 8–10) en fonction du nombre de tâches des instances. L'efficacité de ces bornes est aussi comparée à celle des bornes inférieures obtenues après la propagation initiale (colonnes 2–4). Les temps de calcul sont mentionnés uniquement pour les bornes inférieures destructives renforcées par du *shaving* (colonnes 11–13), parce qu'ils sont négligeables pour les autres bornes ($\ll 1$ second). Remarquez que la présence des contraintes (8.5) contribue grandement à améliorer la qualité de ces bornes inférieures. La règle (AF) n'améliore par la borne inférieure durant la propagation initiale, alors qu'elle améliore significativement toutes les bornes inférieures destructives. Dans tous les cas, la règle (PF) améliore légèrement la qualité des bornes obtenues avec la règle (AF). Par ailleurs, la qualité les bornes inférieures destructives sans *shaving* est satisfaisante tout en conservant un temps de calcul négligeable. Par contre, la phase de *shaving* améliore ces bornes, mais entraîne une

1. http://www.mines-nantes.fr/en/Sites-persos/Christelle-GUERET/Batch-processing-problems

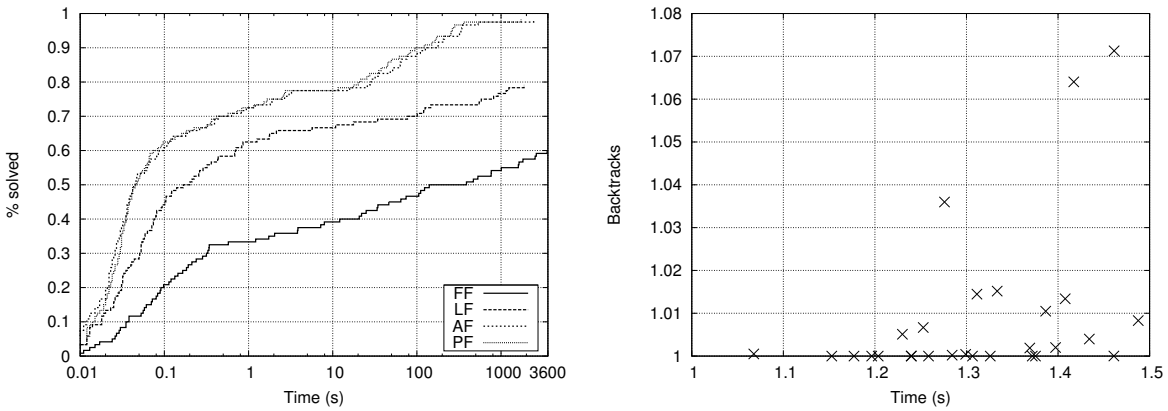
	Initial Propag.			Destr. LB			Destr. LB + Shaving					
n	LF	AF	PF	LF	AF	PF	LF	AF	PF	$\bar{t}_{\mathrm{LF}}$	$\bar{t}_{\mathrm{AF}}$	$\bar{t}_{\mathrm{PF}}$
10	89.4	89.4	89.7	89.4	97.9	98.1	93.4	99.0	**99.2**	0.07	**0.05**	**0.05**
20	90.1	90.1	90.4	90.1	95.6	95.7	93.6	96.8	**97.0**	0.17	0.15	**0.1**
50	89.7	89.7	90.2	89.7	93.6	93.6	91.8	**94.1**	**94.1**	2.71	2.71	**1.43**
75	88.5	88.5	89.1	88.5	91.4	91.5	89.8	91.7	**91.8**	8.8	12.01	**5.25**
100	87.2	87.2	87.6	87.2	89.4	89.4	88.3	89.6	**89.7**	23.20	29.48	**14.04**

TABLE 8.1 – Qualité des bornes inférieures destructives.

augmentation importante du temps de calcul. Remarquez que la règle (PF) améliore la qualité des bornes inférieures destructives avec *shaving* tout en diminuant leurs temps de calcul de moitié.

8.4.1.2 Complete decreasing first fit

La procédure *complete decreasing first fit* [67] place les tâches par ordre décroissant de taille en plaçant chaque tâche dans la première fournée pouvant la contenir. Nous désactivons le mécanisme de calcul des bornes inférieures destructives pour réduire l'influence des règles de filtrage sur la stratégie de branchement. Nous avons sélectionné un sous-ensemble d'instances qui peuvent être résolues optimalement par la plupart des règles de filtrage, depuis (FF) jusqu'à (PF), pour que les comparaisons puissent être réalisées dans un laps de temps raisonnable. Dans ce but, nous avons sélectionné toutes les instances avec moins de 50 tâches (120 instances).
La figure 8.3(a) donne le pourcentage d'instances résolues en fonction du temps (en secondes) pour chacune des règles de filtrage. Tout d'abord, les performances de la règle (FF) montrent que notre modèle est capable de capturer des solutions optimales même lorsque le filtrage de la variable objectif n'a lieu qu'après l'instanciation de toutes les durées et dates échues des fournées. Nous pensons que notre modèle en programmation par contraintes est très efficace pour détecter les placements des jobs menant à des ordonnancements équivalents par rapport à la règle EDD. Ensuite, la règle (LF) basée uniquement sur la relaxation du problème améliore la qualité des solutions et les temps de calcul par rapport à la règle (FF), même si un certain nombre d'instances ne sont pas résolues optimalement. Finalement, seules deux instances avec 50 tâches ne sont pas résolues optimalement dans la limite de temps quand on applique la règle (AF) ou (PF). Par contre, la figure ne montre pas clairement le gain offert par la règle (PF) par rapport à la règle (AF).
Pour répondre à cette question, la figure 8.3(b) compare en détail la résolution des instances avec les règles (AF) et (PF). Chaque point représente une instance dont la coordonnée x est le quotient du temps



(a) Comparaison du temps de résolution ($n \leq 50$). (b) (AF) ÷ (PF) ($n = 50$).

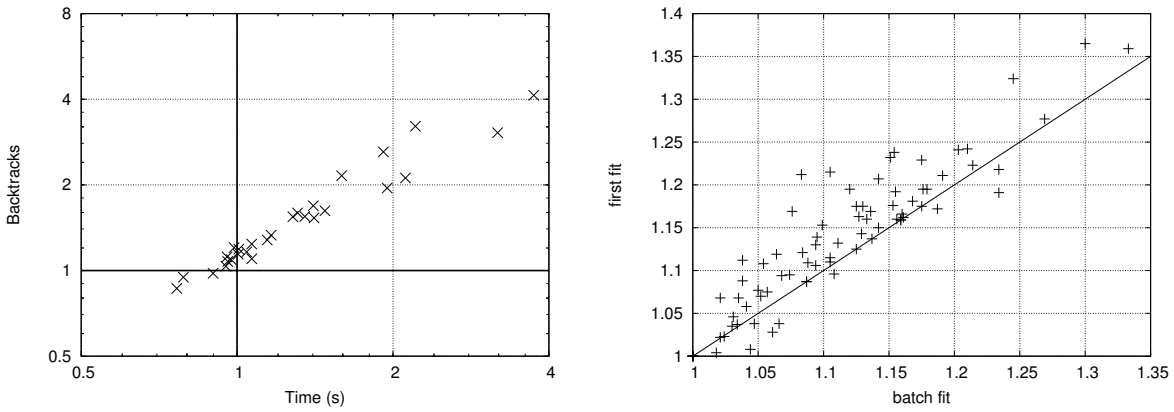
FIGURE 8.3 – Comparaison des règles de filtrage en utilisant *complete decreasing first fit*.

de résolution avec la règle (AF) sur celui avec (PF) et la coordonnée y est le quotient du nombre de backtracks avec la règle (AF) sur celui avec (PF). Nous ne considérons que les instances dont le temps de résolution est supérieur à 2 secondes, ce qui ne se produit que pour celles avec 50 tâches. Tous les points sont situés en haut à droite du point (1,1), car l'utilisation de la règle (PF) améliore la résolution de toutes les instances sans exception. En fait, la règle (PF) a un comportement similaire à celui observé pendant le calcul des bornes inférieures destructives. Son utilisation diminue toujours le temps de résolution et certaines instances sont même résolues approximativement 30 % plus vite, mais le nombre de backtracks est presque identique avec les deux règles.

8.4.2 Performance des heuristiques de sélection de valeur

Nous mesurons maintenant l'influence de l'heuristique de sélection de valeur sur la résolution. Dans ce but, nous résolvons toutes les instances en utilisant toutes les règles de filtrage, mais aucune borne inférieure destructive. Nous comparerons seulement *batch fit* à *first fit*, parce que *first fit* et *best fit* obtiennent des performances équivalentes dans le cadre de notre approche. En effet, la différence entre les temps de résolution avec *first fit* et *best fit* est toujours inférieure à 5 secondes pour les instances avec 50 tâches et cette différence correspond au plus à 2 % du temps de résolution. De plus, les deux heuristiques atteignent toujours la même borne supérieure pour les instances avec plus de 50 tâches.
La figure 8.4(a) étudie les effets de l'heuristique de sélection de valeur pour les instances avec 50 tâches et un temps de résolution supérieur à 2 secondes. Chaque point représente une instance dont la coordonnée x est le quotient du temps de résolution avec *first fit* sur celui avec *batch fit* et la coordonnée y est le quotient du nombre de backtracks avec *first fit* sur celui avec *batch fit*. Les deux axes ont des échelles logarithmiques. Tous les points situés au-dessus ou à droite du point (1,1) représentent des instances dont la résolution est améliorée par l'utilisation de *batch fit* (quadrant en haut à droite). Au contraire, les quelques points situés en dessous ou à gauche du point (1,1) sont des instances pour lesquelles il est préférable de garder *first fit* (quadrant en bas à gauche). Comme espéré, tous les points se situent autour de la diagonale ($x = y$) puisque le nombre de backtracks est à peu près proportionnel au temps de résolution (les quadrants en haut à gauche et en bas à droite sont vides).
La figure 8.4(b) analyse les effets de l'heuristique de sélection de valeur sur les instances avec plus de 50 tâches. Notons lb la meilleure borne supérieure connue pour une instance [2]. La qualité d'une solution ub est estimée en mesurant son écart à lb grâce à une formule inspirée par les travaux de Malve et Uzsoy [135] : $(ub + d_{max}) \div (lb + d_{max})$. Cet écart est généralement supérieur à l'écart à l'optimum puisque ces instances n'ont pas toujours été résolues optimalement. Chaque point représente une instance dont

2. http://www.mines-nantes.fr/en/Sites-persos/Christelle-GUERET/Batch-processing-problems



(a) *first fit* ÷ *batch fit* ($n = 50$).

(b) Comparaison des écart à la borne inférieure ($n > 50$).

FIGURE 8.4 – Comparaison des heuristiques de sélection de valeur.

la coordonnée x est l'écart à la borne inférieure obtenu avec *batch fit* alors que la coordonnée y est celui obtenu avec *first fit*. Tous les points situés au-dessus de la droite $(x = y)$ correspondent à des instances pour lesquelles *batch fit* a retourné une meilleure solution. L'utilisation de *batch fit* améliore globalement la résolution même si les performances sur quelques rares instances sont dégradées.

8.4.3 Comparaison avec un modèle en programmation mathématique

Notre approche est maintenant comparée à une formulation en programmation mathématique inspirée par les travaux de Daste *et al.* [133]. Soit x_{jk} une variable booléenne prenant la valeur 1 si la tâche j est placée dans la k-ème fournée de la séquence. Les variables entières positives P_k, D_k, et C_k représentent respectivement la durée d'exécution, la date échue et la date d'achèvement de la k-ème fournée.

$$\min L_{max} \tag{8.6}$$

Subject to :

$$\sum_{k \in K} x_{jk} = 1 \qquad \forall j \in J \tag{8.7}$$

$$\sum_{j \in J} s_j x_{jk} \leq b \qquad \forall k \in K \tag{8.8}$$

$$p_j x_{jk} \leq P_k \qquad \forall j \in J,\ \forall k \in K \tag{8.9}$$

$$C_{k-1} + P_k = C_k \qquad \forall j \in J,\ \forall k \in K \tag{8.10}$$

$$(d_{max} - d_j)(1 - x_{jk}) + d_j \geq D_k \qquad \forall j \in J,\ \forall k \in K \tag{8.11}$$

$$D_{k-1} \leq D_k \qquad \forall k \in K \tag{8.12}$$

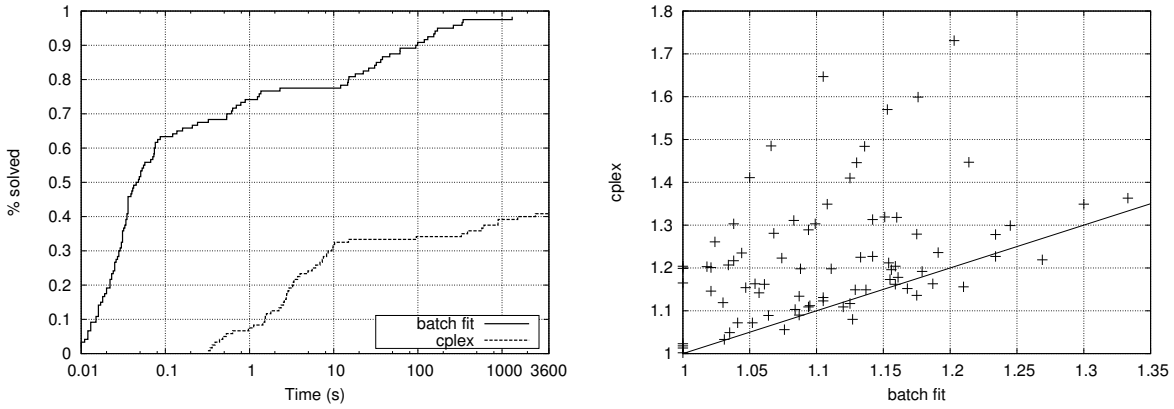
$$C_k - D_k \leq L_{max} \qquad \forall k \in K \tag{8.13}$$

$$\forall j \in J,\ \forall k \in K,\ x_{jk} \in \{0, 1\}$$

$$\forall k \in K,\ C_k \geq 0,\ P_k \geq 0,\ D_k \geq 0$$

Les contraintes (8.6) définissent le critère d'optimalité qui consiste à minimiser le retard algébrique maximal des tâches. Les contraintes (8.7) imposent que chaque tâche soit placée dans exactement une fournée et les contraintes (8.8) imposent que la capacité b de la machine soit respectée dans chaque fournée. Ces contraintes définissent le modèle pour le problème de *bin packing* proposé par Martello et Toth [64]. Les contraintes (8.9) imposent que la durée d'exécution de chaque fournée k soit égale à la durée d'exécution maximale de ses tâches. Les contraintes (8.10) imposent que la date d'achèvement de la k-ème fournée soit égale à la date d'achèvement de la $(k-1)$-ème fournée additionnée à la durée d'exécution de la k-ème fournée, c'est-à-dire qu'une solution est une séquence de fournée sans temps mort en suivant leur numérotation. Il est nécessaire d'ajouter une fournée fictive terminant à l'origine du temps ($C_0 = 0$). Les contraintes (8.11) imposent que la date échue D_k de la k-ème fournée soit égale à la plus petite date échue de ses tâches. Les contraintes (8.12) imposent le respect de la règle EDD par la séquence de fournées. Ces contraintes améliorent la résolution, car les permutations des fournées menant à un ordonnancement équivalent en appliquant la règle EDD sont interdites, mais elles ne sont pas nécessaires à la correction du modèle. Les contraintes (8.13) sont la définition du retard des fournées et imposent que la valeur de l'objectif soit égale au retard algébrique maximal. Finalement, une solution de la formulation en programmation mathématique est un placement réalisable des tâches dans une séquence de fournées respectant la règle EDD. À l'opposé du modèle en programmation par contraintes, cet encodage de la solution ne détecte pas toutes les permutations équivalentes des fournées.

Notre approche applique l'heuristique de sélection de valeur *batch fit* et toutes les règles de filtrage. Les comparaisons ont été réalisées avec la formulation en programmation mathématique implémentée comme un modèle Ilog OPL 6.1.1 et résolue par `Ilog Cplex` 11.2.1. Le solveur `Ilog Cplex` fournit un outil de configuration automatique qui tente de trouver la meilleure combinaison de ses paramètres. Cet outil a été seulement appliqué sur les instances avec 10 tâches, car il est nécessaire de les résoudre optimalement plusieurs fois. Tout d'abord, cet outil a modifié le type de *coupe* pour réduire le nombre de branches explorées. Une *coupe* est une contrainte ajoutée au modèle pour supprimer des solutions optimales fractionnaires de la relaxation continue. Ensuite, cet outil privilégie la satisfiabilité en plaçant la tâche j dans la fournée k dans la première branche ($x_{jk} = 1$) à la place du comportement par défaut qui interdit ce placement ($x_{jk} = 0$).
La figure 8.5(a) montre le pourcentage d'instances résolues optimalement en moins de 3600 secondes (1h)



(a) Comparaison du temps de résolution ($n \leq 50$). (b) Comparaison des écart à la borne inférieure ($n > 50$).

FIGURE 8.5 – Comparaison à la formulation mathématique.

comme une fonction du temps de résolution Notre approche en programmation par contraintes obtient de meilleurs résultats que la formulation mathématique : elle résout un plus grand nombre d'instances avec des temps de résolution inférieurs d'un ordre de grandeur.
La figure 8.5(b) compare les écarts à la meilleure borne inférieure connue pour les instances avec plus de 50 tâches. La limite de temps de `Ilog Cplex` est relevée à 43200 secondes (12h). Chaque point représente une instance dont la coordonnée x est l'écart à la borne inférieure obtenu par notre approche et la coordonnée y est l'écart obtenu par `Ilog Cplex`. Tous les points situés au-dessus de la droite ($x = y$) sont des instances pour lesquelles notre approche a retourné la meilleure solution. Malgré une limite de temps d'une heure, notre approche renvoie une meilleure solution que la formulation mathématique pour la grande majorité des instances. De plus, la différence entre les deux solutions est très faible lorsque la formulation mathématique retourne la meilleure solution, alors qu'elle peut être significative lorsqu'il s'agit de notre approche.

8.4.4 Comparaison avec une approche par branch-and-price

Dans cette section, nous nous comparons à l'approche par branch-and-price de Daste *et al.* [151] décrite en section 5.3. Nous rappelons qu'une solution du problème maître est une séquence réalisable de fournées et que chaque colonne représente une fournée. À chaque itération, on résout un sous-problème pour déterminer une colonne (fournée) qui améliore la solution du problème maître par un algorithme glouton, puis si nécessaire, par une méthode exacte d'énumération. Leur approche a été testée sur le jeu d'instances utilisé présentement avec une limite de temps de 3600 secondes (1h), mais en se limitant aux instances avec moins de 50 tâches. Toutes leurs expérimentations ont été réalisées sur un Pentium avec un processeur cadencé à 2.66 GHz et 1 GB de mémoire vive. Les résultats détaillés sur ces instances ne sont pas précisés. Leur approche par branch-and-price résout respectivement 55% et 8% des instances contenant 20 et 50 tâches. Leurs expérimentations montrent que l'approche par branch-and-price est plus efficace que la formulation mathématique, mais celle-ci reste moins performante que notre approche en programmation par contraintes qui résout toutes les instances contenant 20 tâches en moins d'une seconde et 95% des instances contenant 50 tâches avec un temps moyen de résolution inférieur à 100 secondes.

8.5 Conclusion

Dans ce chapitre, nous avons présenté une approche par programmation par contraintes pour minimiser le retard algébrique maximal d'une machine à traitement par fournées sur laquelle un nombre fini de tâches avec des tailles différentes doivent être ordonnancées. Cette approche exploite une nouvelle

contrainte globale pour l'optimisation basée sur une relaxation du problème dont on déduit des règles de filtrage basées sur les coûts. L'exploration de l'arbre de recherche est améliorée grâce à des heuristiques de sélection de variable et de valeur. Les résultats expérimentaux montrent d'abord l'intérêt des différentes composantes de notre approche. Les comparaisons avec une formulation mathématique et une approche par branch-and-price révèlent que notre approche fournit de meilleures solutions avec des temps de résolution inférieurs d'un ordre de grandeur.
Dans de futurs travaux, nous souhaitons adapter cette approche pour d'autres critères d'optimalité relatifs aux dates d'achèvement (C_{max}, $\sum C_j$, $\sum w_j C_j$). D'autres sujets de recherche concernent le cas à plusieurs machines à traitement par fournées ou la présence de contraintes additionnelles, notamment des dates de disponibilité qui sont actuellement incompatibles avec notre approche.

Annexe 8.A : un algorithme quadratique pour le filtrage du placement des tâches

Nous rappelons qu'un algorithme basique peut calculer indépendamment les coûts marginaux de tous les placements possibles avec une complexité $O(n^3)$. Dans cette section, nous décrivons un algorithme de complexité $O(nm)$ basé sur le calcul incrémental des coûts marginaux nommé jobCBF. L'idée principale est d'exploiter la formule (8.14) pour calculer rapidement les coûts marginaux en distinguant plusieurs cas de placement. Par exemple, tous les placements d'une tâche dans une fournée vide ont le même coût marginal. En fait, la date d'achèvement d'un bloc p après le placement de la tâche j dans la fournée $k \in \mathcal{D}(B_j)$ est la suivante :

$$C_p(\mathcal{A} \cup \{B_j \leftarrow k\}) = \begin{cases} C_p(\mathcal{A}) & \text{if } \delta_p < \min(d_j, \max(D_k)) \quad (8.14a) \\ C_p(\mathcal{A}) + \max(p_j, \min(P_k)) & \text{if } d_j \le \delta_p < \max(D_k) \quad (8.14b) \\ C_p(\mathcal{A}) + \max(p_j - \min(P_k), 0) & \text{if } \delta_p \ge \max(D_k) \quad (8.14c) \end{cases}$$

La figure 8.6 illustre le principe de la formule (8.14) : le placement d'une tâche j dans une fournée k implique son transfert de la fournée k depuis un bloc p vers un de ses prédécesseurs ou lui-même, c'est-à-dire un bloc $p' \le p$ tel que $\delta_{p'} = \min(d_j, \max(D_k))$. Tout d'abord, la séquence de blocs $1, \ldots, p'-1$ n'est pas modifiée par le transfert. Ainsi, les dates d'achèvement et les retards de ces blocs restent identiques. Par conséquent le retard algébrique maximal de cette sous-séquence de blocs est dominé par $L(\mathcal{A})$. En effet, la règle (LF) est appliquée après l'initialisation des blocs, donc avant la règle (AF). Ensuite, les dates d'achèvement des blocs $p', \ldots, p-1$ sont retardées de la durée actualisée $\max(\min(P_k), p_j)$ de la fournée et elles sont donc données par (8.14b). Finalement, l'augmentation de la durée $\max(p_j - \min(P_k), 0)$ de la fournée k est ajoutée aux dates d'achèvement des blocs $p, \ldots, n$ en (8.14c). Si cette augmentation est nulle, alors le retard algébrique maximal des blocs $p, \ldots, n$ est dominé par $L(\mathcal{A})$

L'algorithme réduit le domaine des variables B_j en appliquant la règle (AF). Le filtrage est illustré en figure 8.7 pour le placement de la tâche 4 en partant de l'affectation partielle $\mathcal{A}_2$ (voir figure 8.1 page 81). Après l'initialisation, la boucle L1 parcourt les blocs en ordre inverse pour détecter les placements non réalisables. Le retard algébrique maximal $\Lambda_{\ge p}$ et l'augmentation maximale autorisée de la durée Δ de la sous-séquence de blocs $p, \ldots, n$ sont actualisés grâce à des formules récursives déduites de la section 8.2.1.

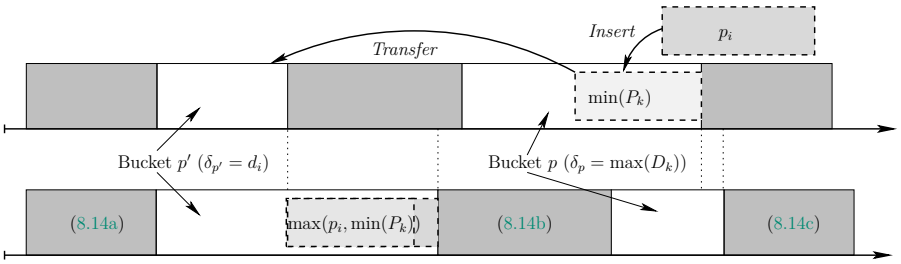
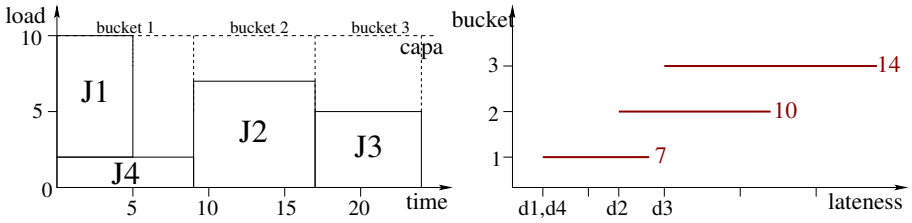


FIGURE 8.6 – Anticipation de l'effet du placement d'une tâche sur la séquence de blocs.

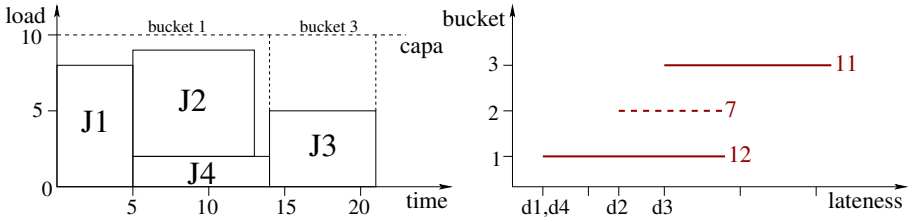
Procédure jobCBF : Filtrage basé sur le coût du placement des tâches en temps quadratique.

$\mathcal{B} = \emptyset$; // Set of batches to prune
$\Lambda_{\geq p} = -\infty$; // Lateness of sub-sequence $p, \ldots, n$
foreach $k \in K$ **do** $\Lambda^k_{\geq p} = -\infty$; // Lateness without batch k of sub-sequence $p, \ldots, n$
/* Try assignments in decreasing order of blocs */
L1 **for** $p \leftarrow n$ **to** 1 **do**
  $\Lambda_{\geq p} = \max(L_p(\mathcal{A}), \Lambda_{\geq p})$; // Maximal lateness of sub-sequence $p, \ldots, n$
  $\Delta = \max(L_{max}) - \Lambda_{\geq p}$; // Maximal duration increase allowed at bloc p
  $\mathcal{B}_{=p} = \{k \in K_{=p} \mid \min(P_k) > 0 \wedge (|\mathcal{D}(P_k)| > 1 \vee |\mathcal{D}(D_k)| > 1)\}$; // New batches to prune
  /* Update durations of new batches at bloc p */
L2 **foreach** $k \in \mathcal{B}_{=p}$ **do** $\max(P_k) \leftarrow \min(P_k) + \Delta$;
  /* assignments of an available job j to other batches */
L3 **foreach** $k \in \mathcal{B}$ **do** $\Lambda^k_{\geq p} = \max(L_p(\mathcal{A}), \Lambda^k_{\geq p})$;
L4 **forall the** $j \in J_{=p}$ **such that** $|\mathcal{D}(B_j)| > 1$ **do**
    /* Assignment to non-empty batches */
L5 **foreach** $k \in \mathcal{B}$ **do**
      if $\delta_p < \min(D_k)$ **then** $\mathcal{B} = \mathcal{B} \backslash \{k\}$;
      else if $\Lambda^k_{\geq p} + \max(\min(P_k), p_j) > \max(L_{max})$ **then** $B_j \not\leftarrow k$;
    /* Assignment to empty batches */
    if $p_j > \Delta$ **then** $\max(B_j) \leftarrow |K^\star|$;
  /* Update data structure */
L6 **foreach** $k \in \mathcal{B}_{=p}$ **do** $\Lambda^k_{\geq p} = \Lambda_{\geq p} - \min(P_k)$;
  $\mathcal{B} = \mathcal{B} \cup \mathcal{B}_{=p}$;

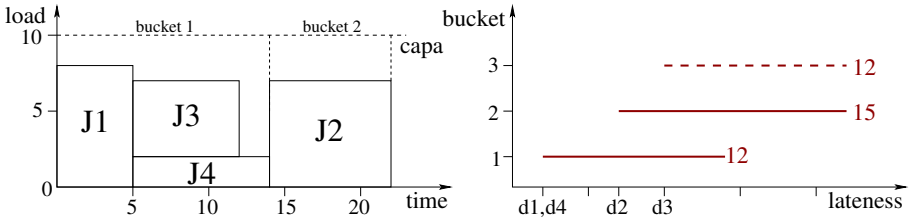
Les fournées de $\mathcal{B}_{=p}$ ordonnancées au plus tard dans le bloc p qui ne sont ni vides ni pleines, sont maintenant examinées par l'algorithme. La boucle L2 actualise la durée maximale autorisée de ces fournées, mais la suppression des placements non réalisables est déléguée aux contraintes (8.1). Cette boucle détecte simultanément tous les placements non réalisables d'une tâche j dans une fournée k tels que $\max(D_k) \leq d_j$. En effet, leurs coûts marginaux ne dépendent que de l'augmentation de la durée de la fournée, car (8.14b) n'est pas défini. Ce cas de figure est illustré en figure 8.7(a).
La boucle L3 actualise incrémentalement le retard algébrique maximal $\Lambda^k_{\geq p}$ de la séquence $p, \ldots, n$ sans la fournée k. Puis, la boucle L4 examine seulement les placements de tâches qui entraînent effectivement le transfert d'une fournée vers le bloc p. La boucle interne L5 examine donc les fournées découvertes dans des blocs précédemment visités par l'algorithme. Si le transfert de la fournée dans le bloc courant n'est pas réalisable à cause des contraintes portant sur les dates échues, alors la fournée est supprimée de l'ensemble $\mathcal{B}$ des fournées à examiner. Dans le cas contraire, le placement de la tâche j dans la fournée k est éliminé lorsque le retard algébrique maximal après le transfert de la fournée dans le bloc p dépasse la meilleure borne supérieure connue. En fait, le calcul incrémental de $\Lambda^k_{\geq p}$ est crucial dans cet algorithme, car il réduit la complexité de la formule (8.14b) d'un temps linéaire à un temps constant. Dans la figure 8.7(b), le transfert de la fournée 2 depuis le bloc 2 vers le bloc 1 ne change pas la séquence de fournées dans la solution de la relaxation, alors que le transfert de la fournée 3 du bloc 3 vers le bloc 1 inverse les positions des fournées 2 et 3 dans la figure 8.7(c).
Ensuite, l'algorithme examine simultanément tous les placements d'une tâche dans une fournée vide pour lesquels les formules (8.14b) and (8.14c) sont égales, car l'augmentation de la durée de la fournée est égale à sa durée ($\min(P_k) = 0$) comme illustré en figure 8.7(d). Dans ce cas, le domaine de la variable B_j est réduit aux valeurs inférieures à l'indice $|K^\star|$ de la dernière fournée non vide.
Finalement, la boucle L6 initialise les retards algébriques maximaux $\Lambda^k_{\geq p}$ des fournées du bloc p en soustrayant leur contribution au retard de la sous-séquence. Ensuite, l'ensemble $\mathcal{B}$ des fournées à examiner est mis à jour.
La complexité de la procédure dépend uniquement de ces boucles puisque toutes les instructions sont exécutées en temps constant. Un calcul simple montre que la complexité ne dépend que des boucles imbriquées L1, L4 et L5. Dans le pire cas, les boucles imbriquées L1 et L4 parcourent une partition de l'ensemble des tâches J. Ce parcours a une complexité $O(n)$. La boucle L5 est réalisée en $O(m)$, car $\mathcal{B} \subseteq K$ est un sous-ensemble de fournées. Par conséquent, la complexité de l'algorithme est en $O(nm)$.



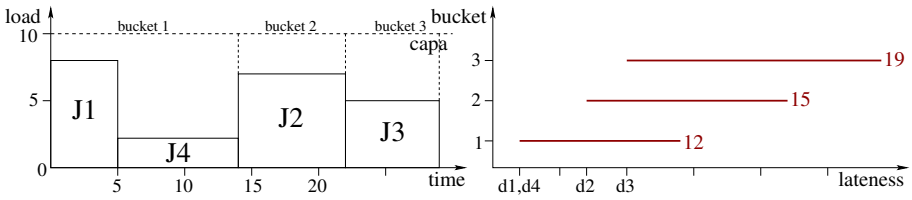
(a) Placement : $B_4 \leftarrow 1$



(b) Placement : $B_4 \leftarrow 2$



(c) Placement : $B_4 \leftarrow 3$



(d) Placement : $B_4 \leftarrow 4$

FIGURE 8.7 – Solution de la relaxation après le placement de la tâche 4 dans une fournée pour l'affectation partielle $\mathcal{A}_2 = \{B_1 \leftarrow 1, B_2 \leftarrow 2, B_3 \leftarrow 3, B_4 \in [1, 4]\}$.

Chapitre 9

Implémentation dans le solveur de contraintes choco

Nous présentons nos principales contributions au solveur de contraintes choco*. La plupart concernent le développement d'un module d'ordonnancement. Nous évoquerons brièvement d'autres contributions sur les problèmes de placement et certains mécanismes internes du solveur. Finalement, nous examinerons les résultats obtenus sur les problèmes d'ordonnancement par* choco *lors d'une compétition de solveurs. En complément, les annexes A et B décrivent respectivement des outils de développement et d'expérimentation et plusieurs cas d'utilisation.*

Sommaire

9.1 Le modèle . . . 94
9.1.1 Variables . . . 94
9.1.2 Contraintes . . . 95
9.2 Le solveur . . . 96
9.2.1 Lecture du modèle . . . 96
9.2.2 Stratégie de branchement . . . 97
9.2.3 Redémarrages . . . 98
9.2.4 Résolution d'un modèle . . . 99
9.3 Contraintes de choco . . . 99
9.3.1 cumulative . . . 100
9.3.2 disjunctive . . . 101
9.3.3 useResources (en développement) . . . 102
9.3.4 precedence . . . 103
9.3.5 precedenceDisjoint . . . 103
9.3.6 precedenceReified . . . 103
9.3.7 precedenceImplied . . . 104
9.3.8 disjoint (décomposition) . . . 104
9.3.9 pack . . . 104
9.3.10 Reformulation de contraintes temporelles . . . 106
9.4 Construction du modèle disjonctif . . . 106
9.4.1 Traitement des contraintes temporelles . . . 107
9.4.2 Traitement des contraintes de partage de ressource . . . 107
9.4.3 Déduction de contraintes « coupe-cycle » . . . 109
9.5 CSP Solver Competition 2009 . . . 110
9.5.1 Catégorie 2-ARY-INT . . . 110
9.5.2 Catégorie Alldiff+Cumul+Elt+WSum . . . 113
9.6 Conclusion . . . 113

Le succès de la programmation par contraintes est dû à la simplicité de la modélisation grâce à un langage déclaratif de haut niveau, mais aussi à la disponibilité de solveurs de contraintes efficaces. L'architecture, l'implémentation et l'évaluation de ces systèmes relèvent encore du domaine de la re-

cherche [171] tant il est difficile de trouver un équilibre entre la simplicité, l'efficacité et la flexibilité. Par conséquent, nous discutons dans ce chapitre de nos principales contributions au solveur choco, évaluées dans les chapitres précédents et à la fin de celui-ci. Nous insisterons sur le langage déclaratif proposé pour l'ordonnancement sous contraintes, et nous évoquerons succinctement certains choix d'architecture et d'implémentation.

Le solveur choco est une librairie Java pour la programmation par contraintes. Il est basé sur un moteur de propagation réagissant à des évènements et des structures de données réversibles. Le solveur choco est un logiciel libre distribué sous une **licence BSD** et hébergé sur sourceforge.net.

L'organisation de ce chapitre est la suivante. Les sections 9.1 et 9.2 introduisent les deux éléments essentiels de choco que sont le `Model` et le `Solver` à travers le prisme de l'ordonnancement sous contraintes. La section 9.3 détaille les principales contraintes implémentées durant cette thèse et maintenant disponibles dans choco. La section 9.4 présente une procédure de construction automatique du graphe disjonctif généralisé (voir section 3.4) à partir d'un modèle de base défini par l'utilisateur. Finalement, la section 9.5 récapitule les résultats de l'édition 2009 de la *CSP solver competition* sur les problèmes d'ordonnancement. Ce chapitre est axé sur l'ordonnancement sous contraintes et nous invitons le lecteur à consulter http://choco.mines-nantes.fr pour de plus amples informations. La documentation officielle est disponible à cet adresse.

9.1 Le modèle

Le `Model`, accompagné du `Solver`, est l'un des éléments fondamentaux de choco. Il permet de décrire simplement un problème de manière déclarative. Son rôle principal est d'enregistrer les variables et contraintes d'un modèle. Nous introduisons dans cette section les variables et contraintes utiles à la modélisation de problèmes d'ordonnancement ou de placement en une dimension.
On accède généralement à l'API pour construire un `Model` par un *import statique*.

```
1  import static choco.Choco.*;
   Model model = new CPModel(); //Modèle PPC
```

9.1.1 Variables

Une variable est définie par son type (entier, réel, ensemble), un nom et les valeurs de son domaine. On peut spécifier certaines options pour imposer une représentation du domaine dans le `Solver`, par exemple un domaine borné ou énuméré pour une variable entière. Un choix judicieux des types des domaines en fonction de leur taille et des contraintes du modèle portant sur leur variable a une influence critique sur l'efficacité du solveur. Une combinaison de variables entières par des opérateurs (arithmétiques et logiques) est représentée par la classe `IntegerExpressionVariable`. On ajoute explicitement des variables au modèle grâce aux méthodes :

```
1  addVariable(String options, Variable v);
   addVariables(String options, Variable... v);
```

Nous rappelons la signification de deux options pour les variables entières pour l'optimisation et l'ordonnancement :
`Options.V_OBJECTIVE` déclare une `Variable` comme objectif (optimisation).
`Options.V_MAKESPAN` déclare une `IntegerVariable` comme la date d'achèvement de l'ordonnancement, c'est-à-dire de *toutes* les tâches du solveur.
L'interface de création des variables simples est détaillée dans la documentation officielle.

La classe `MultipleVariables` représente une variable complexe, c'est-à-dire un conteneur d'objets `Variable` à qui sont associés une sémantique. Une tâche non préemptive définie en section 3.1 est représentée par un objet `TaskVariable` héritant de la classe `MultipleVariables`. Sa sémantique est définie dans l'interface `ITaskVariable` qui donne accès à ces variables entières (`IntegerVariable`) : sa date de début (`start`), sa date de fin (`end`) et sa durée positive (`duration`). Une tâche non préemptive est dite consistante si ses variables entières vérifient la relation : `start` + `duration` = `end`.
Nous avons justifié empiriquement en section 3.1 le choix d'utiliser trois variables et non deux (par

exemple `start` et `duration`). Une limitation de choco réduit encore l'intérêt d'utiliser une représentation à deux variables. En effet, la modélisation de la troisième variable par une expression arithmétique (`IntegerExpressionVariable`) est insuffisante, car ces objets ne peuvent pas appartenir au scope d'une contrainte globale. Dans ce cas, un utilisateur doit, lui-même, définir, gérer l'unicité et assurer la consistance de la troisième variable ce qui réduit la lisibilité du modèle et augmente le risque d'erreur. Le tableau 9.1 décrit l'API pour la création d'une ou plusieurs `TaskVariable` en précisant explicitement les variables entières ou les fenêtres de temps des tâches. *Nous supposons que l'origine des temps est l'instant $t = 0$ et que les durées des tâches sont positives. Il n'y a aucune garantie sur la résolution d'un modèle ne respectant pas ces hypothèses.*

return type : `TaskVariable`
`makeTaskVar(String name, IntegerVariable start, IntegerVariable end, IntegerVariable duration, String... options)`
`makeTaskVar(String name, IntegerVariable start, IntegerVariable duration, String... options)`
`makeTaskVar(String name, int binf, int bsup, IntegerVariable duration, String... options)`
`makeTaskVar(String name, int binf, int bsup, int duration, String... options)`
`makeTaskVar(String name, int bsup, IntegerVariable duration, String... options)`
`makeTaskVar(String name, int bsup, int duration, String... options)`
return type : `TaskVariable[]`
`makeTaskVarArray(String prefix, IntegerVariable[] starts, IntegerVariable[] ends, IntegerVariable[] durations, String... options)`
`makeTaskVarArray(String name, int binf[], int bsup[], IntegerVariable[] durations, String... options)`
`makeTaskVarArray(String name,int binf, int bsup, IntegerVariable[] durations, String... options)`
`makeTaskVarArray(String name, int binf, int bsup, int[] durations, String... options)`
return type : `TaskVariable[][]`
`makeTaskVarArray(String name, int binf, int bsup, IntegerVariable[][] durations, String... options)`
`makeTaskVarArray(String name, int binf, int bsup, int[][] durations, String... options)`

TABLE 9.1 – Méthodes pour la création d'une ou plusieurs tâches (`TaskVariable`).

9.1.2 Contraintes

Pour un solveur non commercial, choco propose un catalogue étoffé de contraintes simples et globales. Une contrainte porte sur une ou plusieurs variables et restreint les valeurs qu'elles peuvent prendre simultanément en fonction d'une relation logique. Le scope d'une contrainte simple contient un nombre prédéterminé de variables contrairement à celui des contraintes globales. Une contrainte globale représente un sous-problème pour lequel elle réalise généralement des réductions de domaine supplémentaires par rapport à un modèle décomposé logiquement équivalent. De plus, l'utilisateur peut facilement définir de nouvelles contraintes génériques ou dédiées à un problème. L'API offre différentes classes abstraites dont peuvent hériter les contraintes utilisateurs.
On poste une contrainte dans le modèle par le biais d'une des méthodes de la classe `Model` :

```
1 addConstraint(Constraint c);
  addConstraints(String options, Constraint... c);
```

L'ajout d'une contrainte au modèle ajoute automatiquement toutes ses variables, c'est-à-dire il n'est pas nécessaire de les ajouter explicitement. Les *options* disponibles pour une contrainte dépendent de son type et déterminent, par exemple, le ou les algorithmes de filtrage appliqués pendant la propagation.

Au début de cette thèse, la notion de tâche n'existait pas et la seule contrainte d'ordonnancement était une contrainte `cumulativeMax` définie sur des variables entières et avec des hauteurs positives. Nous

donnons ici une liste des principales contraintes implémentées dans le cadre de cette thèse :
Temporelles precedence, precedenceDisjoint, precedenceReified, precedenceImplied, disjoint.
Partage de ressource cumulative, disjunctive.
Allocation de ressource useResources.
Sac à dos pack.
Ensemblistes min, max (voir documentation officielle).
La section 9.3 donne une description détaillée des APIs et algorithmes de filtrage.
La liste des contraintes actuellement disponibles en choco est accessible par la javadoc. Certaines contraintes développées parallèlement par d'autres contributeurs traitent de problématiques voisines. On peut citer les contraintes equation et knapsackProblem [73] pour les problèmes de sac à dos, la contrainte geost [172] pour les problèmes de partage de ressource ou de placement géométrique et enfin regular, costRegular [173] et multiCostRegular [174] pour les problèmes d'emploi du temps.

9.2 Le solveur

Les fonctionnalités principales offertes par l'interface Solver sont la lecture du Model et la configuration d'un algorithme de recherche.

9.2.1 Lecture du modèle

Il est fortement recommandé de lire le modèle une seule fois après sa définition complète. La lecture du modèle est décomposée en deux étapes, la lecture des variables et la lecture des contraintes. La création d'un Solver PPC et la lecture d'un modèle sont réalisées de la manière suivante.

```
1 Solver solver = new CPSolver();
  solver.read(model);
```

9.2.1.1 Lecture des variables

Le solveur parcourt les variables du modèle et construit les variables et domaines spécifiques du solveur. Le solveur utilise trois types de variables simples, les variables entières (IntVar), ensemblistes (SetVar) et réelles (RealVar). Le type de domaine est déterminé en fonction des options ou automatiquement.

Les variables du modèle et du solveur sont des entités distinctes. Les variables du solveur sont une représentation des variables du modèle. Ainsi, la notion de valeur d'une variable est définie uniquement dans le solveur. On accède à la variable du solveur à partir d'une variable du modèle grâce à la méthode du Solver : getVar(Variable v). Par exemple, on accède à une tâche du solveur par la méthode getVar(TaskVariable v) qui renvoie un objet TaskVar.

```
1   Model model = new CPModel();
    TaskVariable t = makeTaskVat("T", 0, 100, 5); // model variable
    model.addvariable(t);
    Solver solver = new CPSolver();
5   solver.read(model);
    TaskVar tOnSolver = solver.getVar(x); // solver variable
```

La classe TaskVar implémente les interfaces suivantes :
- Var donne accès au réseau de contraintes et à l'état de la variable.
- ITaskVariable donne accès aux variables entières de la tâche.
- ITask donne accès à la durée et la fenêtre de temps de la tâche.

Le tableau 9.2 donne un aperçu des méthodes publiques de la classe TaskVar.

9.2.1.2 Lecture des contraintes

Après la lecture des variables, le solveur parcourt les contraintes du modèle pour créer les contraintes du solveur qui encapsulent des algorithmes de filtrage réagissant à différents évènements sur les variables

Méthode	Description
`isInstantiated()`	indique si la tâche est ordonnancée (`Var`)
`start()`, `end()`, `duration()`	renvoient ses variables entières (`ITaskVariable`)
`getEST()`, `getLST()`, `getECT()`, `getLCT()`	renvoient les informations sur la fenêtre de temps (`ITask`)
`getMinDuration()`, `getMaxDuration()`	renvoient les informations sur la durée (`ITask`)

TABLE 9.2 – Aperçu des méthodes publiques de la classe `TaskVar`.

(modifications des domaines) ou du solveur (propagation). Les contraintes du solveur sont ajoutées à son réseau de contraintes.

La consistance de tâche est assurée si la tâche appartient au moins à une contrainte de partage de ressource (`cumulative` ou `disjunctive`). Dans le cas contraire, le solveur ajoute automatiquement une contrainte arithmétique assurant la consistance de tâche.
La variable représentant le `makespan` peut être définie par l'utilisateur ou créée automatiquement si nécessaire (en présence de contraintes de partage de ressource). La consistance du `makespan` est assurée par l'ajout automatique de la contrainte : $\texttt{makespan} = \max_{\texttt{tasks}}(\texttt{tasks}[i].\texttt{end})$ où `tasks` contient toutes les tâches du solveur.

L'option `Options.S_MULTIPLE_READINGS` du `Solver` indique que le modèle est lu plusieurs fois. Cette situation se produit par exemple quand on veut propager itérativement des contraintes. Dans ce cas, les contraintes de consistance de tâche et du `makespan` doivent être ajoutées explicitement grâce à un unique appel aux méthodes `CPSolver.postTaskConsistencyConstraints()` et `CPsolver.postMakespanConstraint()`.

9.2.2 Stratégie de branchement

Un ingrédient essentiel d'une approche PPC est une stratégie de branchement adaptée. Au cours de cette thèse, nous avons longuement discuté des algorithmes de recherche et stratégies de branchement utilisés pour résoudre les problèmes d'atelier, et plus généralement les problèmes d'ordonnancement disjonctif, mais aussi des problèmes de fournées et placement. En `choco`, la construction de l'arbre de recherche repose sur une séquence d'objets de branchement qui jouent le rôle des *goals* intermédiaires en programmation logique. L'utilisateur peut définir la séquence d'objets de branchement utilisée lors de la construction de l'arbre de recherche. Outre son large catalogue de contraintes, `choco` propose de nombreuses stratégies de branchement et heuristiques de sélection pour tous les types de variables simples (entière, tâche, réelle) et complexes (tâches). Un axe important de son développement consiste à élaborer des interfaces et classes abstraites pour que les utilisateurs soient capables de développer rapidement leurs propres stratégies et heuristiques. Ces points sont abondamment discutés dans la documentation et nous ne reviendrons pas dessus.

À l'exception notable de la stratégie de branchement de Brucker *et al.* [104] pour les problèmes d'atelier, toutes les stratégies de branchement et heuristiques de sélection présentées dans cette thèse sont maintenant disponibles dans `choco`.

Par souci de simplicité et de concision, nous ne rentrerons pas dans les détails de leur implémentation mais nous mentionnerons néanmoins certains aspects importants. Nos principales contributions concernent la stratégie de branchement *setTimes*, l'heuristique de sélection de disjonction *profile* introduites en section 6.1.4 ainsi que toutes les heuristiques basées sur les degrés (voir sections 2.3 et 7.1.2). Nous proposons des heuristiques de sélection de disjonction et d'arbitrage basées sur la fonction $preserved(x \leq y)$ entre deux variables temporelles proposée par Laborie [122]. Ces dernières n'ont pas été évaluées pendant cette thèse, car nous avons eu cette idée pendant la rédaction du manuscrit.
Plusieurs heuristiques d'arbitrage basées sur l'interface `OrderingValSelector` sont compatibles avec toutes les heuristiques de sélection de disjonction : *minVal*, *random*, *centroid*, *minPreserved* et *maxPreserved*.
Le constructeur de la classe `SetTimes` accepte un objet implémentant l'interface `TaskVarSelector` ou `Comparator<ITask>` qui détermine l'ordre d'exploration des branches (on ordonnance au plus tôt une tâche différente dans chaque branche).
Les heuristiques basées sur les domaines et les degrés sont nombreuses : $(dom \mid slack \mid preserved)/(deg$

| *ddeg* | *wdeg*). Pour chacune, nous proposons le choix entre une stratégie de branchement par *standard labeling* binaire ($x = v \vee x \neq v$) ou n-aire (une valeur différente dans chaque branche) qui influence l'apprentissage des poids. Ainsi, leur implémentation est relativement complexe. Des résolutions successives d'un même modèle par ces heuristiques sont d'ailleurs un test intéressant pour vérifier le déterminisme d'un solveur. Leur implémentation repose sur l'observation suivant : l'évaluation de la condition logique $dom_1 \div deg_1 < dom_2 \div deg_2$ est beaucoup plus lente que celle de la condition équivalente $dom_1 \times deg_2 < dom_2 \times deg_1$. L'implémentation évite soigneusement cet écueil en utilisant des objets héritant de la classe abstraite `AbstractIntVarRatioSelector`. Ensuite, à chaque point de choix, le calcul du degré pondéré de toutes les variables non instanciées est très coûteux. Nous proposons encore des variantes pour les heuristiques (*dom* | *slack* | *preserved*)/*wdeg* où le calcul du degré pondéré est incrémental, approximatif (centré sur les décisions de branchement) et encore expérimental. Le formalisme autour de ces variantes n'est pas complètement défini mais elles obtiennent de très bons résultats expérimentaux.

La classe `BranchingFactory` facilite l'utilisation de toutes ces heuristiques en proposant des méthodes pour la construction d'objets de branchement (par défaut ou personnalisé) sans se soucier des objets intermédiaires. La lecture de son code source constitue une base d'exemples pour mettre en place ses propres « recettes ».

9.2.3 Redémarrages

Nous montrons comment régler les redémarrages et enregistrer des *nogoods* (voir section 6.2.3). Les redémarrages sont intéressants quand la stratégie de recherche intègre des éléments d'apprentissage (*wdeg* et *impact*), ou de randomisation. Les méthodes suivantes appartiennent à la classe `Solver`.
On actionne les redémarrages après la découverte d'une solution (optimisation) :

```
setRestart(boolean restart);
```

On actionne la stratégie de Walsh :

```
setGeometricRestart(int base, double grow);
setGeometricRestart(int base, double grow, int restartLimit);
```

On actionne la stratégie de Luby :

```
setLubyRestart(int base);
setLubyRestart(int base, int grow);
setLubyRestart(int base, int grow, int restartLimit);
```

Le paramètre `base` est le facteur d'échelle, `grow` est le facteur géométrique et `restartLimit` est une limite sur le nombre de redémarrages. On peut configurer ces paramètres en passant directement par l'objet `Configuration` (héritant de la classe `Properties`) du `Solver`.
Par défaut, la longueur d'une exécution dépend du nombre de backtracks mais il est possible d'utiliser une autre mesure (limite, temps, nœuds ...) en modifiant directement la valeur de la propriété `Configuration.RESTART_POLICY_LIMIT`.
Le listing suivant donne un exemple d'activation des redémarrages :

```
  CPSolver s = new CPSolver();
  s.read(model);
  s.setLubyRestart(50, 2, 100);
  //s.setGeometricRestart(14, 1.5d);
       s.setFirstSolution(true);
  s.attachGoal(BranchingFactory.domWDeg(s));
  s.generateSearchStrategy();
  s.launch();
```

L'enregistrement des *nogoods* lors d'un redémarrage est déclenché en modifiant la valeur de la propriété `Configuration.NOGOOD_RECORDING_FROM_RESTART` ou par un appel à la méthode :

```
setRecordNogoodFromRestart(boolean recordNogoodFromRestart);
```

À l'heure actuelle, l'enregistrement des *nogoods* est limité aux branchements portant sur des variables booléennes. Plus précisément, la dernière branche explorée avant le redémarrage est tronquée au premier point de choix portant sur une variable non booléenne. Supposons que notre stratégie de branchement soit composée de deux objets de branchement B_{bool} et B_{int} portant respectivement sur des variables booléennes et entières. Il est possible d'enregistrer des *nogoods* pour la séquence de branchement B_{bool}, B_{int} mais pas pour la séquence B_{int}, B_{bool}.

Notes sur l'implémentation des redémarrages et des nogoods Les *nogoods* sont enregistrés avant un redémarrage. Pour que les *nogoods* soient complets, il est souhaitable que les redémarrages aient lieu avant la création d'un nouveau nœud. Cependant, le mécanisme doit rester assez flexible pour admettre des requêtes de redémarrage provenant de sources diverses. Cette étape a engendré deux modifications majeures du mécanisme interne de `choco` : (a) la boucle de recherche (b) la gestion des limites. Un redémarrage est un mouvement dans la boucle de recherche. On peut le déclencher brutalement en (a) modifiant directement le champ public représentant le prochain mouvement ou (b) déclenchant une `ContradictionException` particulière. Cependant, il est préférable de définir sa propre stratégie (universelle) grâce à l'interface `UniversalRestartStrategy`. Le gestionnaire de limites applique alors cette stratégie de manière sûre et efficace.

9.2.4 Résolution d'un modèle

Les méthodes du tableau 9.3 lancent la résolution d'un problème. Historiquement, `choco` a toujours utilisé une procédure d'optimisation top-down (voir section 2.4). Nous proposons depuis peu d'autres procédures d'optimisation (bottom-up et destructive-lower-bound) et un mécanisme basique de *shaving* (voir section 2.2.4). Ces nouvelles procédures peuvent être activées dans l'objet `Configuration`. Cependant, leur implémentation n'est pas aussi éprouvée que celle de top-down.

Méthode `Solver`	Description
`solve()`	calcule la première solution du modèle, si le modèle est satisfiable.
`solveAll()`	calcule toutes les solutions du modèle, si le modèle est satisfiable.
`minimize(Var obj, boolean restart)`	calcule une solution minimisant la variable objectif `obj`.
`maximize(Var obj, boolean restart)`	calcule une solution maximisant la variable objectif `obj`.
`isFeasible()`	indique le statut du modèle, c'est-à-dire si le modèle est satisfiable, insatisfiable ou si la résolution n'a pas commencé.
`propagate()`	propage les contraintes du modèle jusqu'à ce qu'un point fixe soit atteint ou déclenche une `ContradictionException` si une inconsistance est détectée.

TABLE 9.3 – Méthodes de résolution du `Solver`.

9.3 Contraintes de choco

Dans cette section, nous donnons une description détaillée des contraintes intégrées dans `choco` au cours de cette thèse. La contrainte `sequenceEDD` définie au chapitre 8 n'est pas livrée dans `choco`, car nous la jugeons trop dédiée à notre problème de fournées.

L'implémentation des contraintes de partage de ressource `cumulative` et `disjunctive` repose sur la classe abstraite `AbstractResourceSConstraint` qui définit l'ordre des variables et assure la consistance de tâche. Une tâche `TaskVar` est représentée dans une contrainte de partage de ressource par un objet implémentant l'interface `IRTask` qui donne accès aux autres variables associées à l'exécution de la tâche

sur la ressource (hauteur, usage) et permet de modifier leur domaine de manière transparente, c'est-à-dire sans se soucier de la contrainte représentant la ressource (gestion du réseau de contraintes et de la propagation). Nous rappelons que toutes les tâches d'une ressource régulière sont obligatoires alors que certaines tâches sont optionnelles sur une ressource alternative. Une tâche optionnelle est dite *effective* lorsque la ressource lui a été allouée (une tâche obligatoire est toujours effective).

L'implémentation des contraintes temporelles repose sur la classe abstraite `AbstractPrecedenceSConstraint` qui implémente l'interface `ITemporalRelation`. Elle impose un ordre des variables ainsi qu'une gestion commune des évènements lors de la propagation. Les heuristiques de sélection basées sur le degré pondéré acceptent indifféremment des variables entières ou des objets implémentant l'interface `ITemporalRelation`. Des contraintes temporelles dont le scope ne contient que des variables entières aussi conservées pour assurer la compatibilité avec les versions antérieures de `choco`.

Un module de visualisation des contraintes est disponible par le biais de la fabrique `ChocoChartFactory`. Ce module est basé sur la librairie `jfreechart` pour visualiser les contraintes `cumulative`, `disjunctive` et `pack`. La construction du graphe disjonctif généralisé (voir section 9.4) permet une visualisation du sous-réseau des contraintes `precedence` et `precedenceDisjoint` basée sur la libraire `Graphviz`.

9.3.1 cumulative

Signature `cumulative(String name, TaskVariable[] tasks,IntegerVariable[] heights, IntegerVariable[] usages, IntegerVariable consumption, IntegerVariable capacity, IntegerVariable uppBound, String ... options)`
`tasks` sont les tâches exécutées sur la ressource (recommandé $n \geq 3$).
`heights` sont les hauteurs des tâches . Les hauteurs négatives sont autorisées (producteur-consommateur)
`usages` indiquent si les tâches sont *effectives* sur la ressource (booléens).
`consumption` est la consommation minimale requise sur la ressource.
`capacity` est la capacité de la ressource (recommandé ≥ 2).
`uppBound` est l'horizon de planification pour la ressource (obligatoire ≥ 0).

Description Nous modifions légèrement la définition de la fonction de Dirac $\delta_i(t)$ des moments $\mathcal{M}_i$ d'une tâche préemptive $\texttt{tasks}[i]$ introduite en section 3.1 pour refléter son exécution *effective* sur la ressource :

$$\delta_i(t) = \begin{cases} \texttt{usages}[i] & \text{si } \texttt{tasks}[i].\texttt{start} \leq t < \texttt{tasks}[i].\texttt{end} \\ 0 & \text{sinon} \end{cases}$$

La contrainte `cumulative` impose que la hauteur totale des tâches en cours d'exécution à chaque instant t soit inférieure à la capacité de la ressource (9.1) et supérieure à la consommation minimale lorsqu'au moins une tâche s'exécute (9.2). Elle assure aussi la consistance des tâches non préemptives (9.3). Finalement, elle vérifie la compatibilité des dates de fin des tâches effectives avec son horizon (9.4).

$$\sum_{\texttt{tasks}} \delta_i(t) \times \texttt{heights}[i] \leq \texttt{capacity} \quad (\forall \text{ instant } t) \tag{9.1}$$

$$\sum_{\texttt{tasks}} \delta_i(t) \times \texttt{heights}[i] \geq \texttt{consumption} \times \max_{\texttt{tasks}}(\delta_i(t)) \quad (\forall \text{ instant } t) \tag{9.2}$$

$$\texttt{tasks}[i].\texttt{start} + \texttt{tasks}[i].\texttt{duration} = \texttt{tasks}[i].\texttt{end} \quad (\forall \text{ tâche } i) \tag{9.3}$$

$$\texttt{usages}[i] \times \texttt{tasks}[i].\texttt{end} \leq \texttt{uppBound} \quad (\forall \text{ tâche } i) \tag{9.4}$$

Algorithme de filtrage La section 3.3.2 décrit plus précisément les algorithmes de filtrage évoqués ci-dessous. L'algorithme de filtrage par défaut (on ne peut pas le désactiver) est une implémentation fidèle de l'algorithme à balayage proposé par Beldiceanu et Carlsson [54] qui gère les hauteurs négatives et les tâches optionnelles. Cependant, nous supprimons les règles de filtrage associées à leur hypothèse sur l'allocation de ressource : il existe plusieurs ressources cumulatives et chaque tâche doit être exécutée sur exactement une de ces ressources. En `choco`, l'allocation des ressources est gérée par des contraintes `useResources`.

Lorsque les hauteurs sont positives, les options permettent d'appliquer des algorithmes de filtrage supplémentaires (*task intervals* ou *edge finding*) pour la satisfaction de la contrainte de capacité (9.1). Actuellement, ces algorithmes supplémentaires ne raisonnent pas sur la contrainte de consommation minimale et ignorent purement et simplement les tâches optionnelles.

Options Les options sont des champs (`String`) de la classe `choco.Options`.
`NO_OPTION` filtrage par défaut uniquement : algorithme à balayage [54]
`C_CUMUL_TI` implémentation « maison » des raisonnements énergétiques (*task intervals* ou *overload checking*) en $O(n \log n)$ basée sur une variante des $\Theta - tree$ de Vìlim.
`C_CUMUL_STI` implémentation classique des raisonnements énergétiques en $O(n^2)$. Les inférences sont plus fortes que précédemment puisque le calcul des énergies intègre les tâches à cheval sur un intervalle de temps (sur la droite).
`C_CUMUL_EF` implémentation de l'*edge finding* en $O(n^2)$ basée sur l'algorithme CalcEF proposé par Mercier et Hentenryck [56] (le calcul de R est paresseux).
`C_CUMUL_VEF` implémentation de l'*edge finding* (*en développement*) pour une ressource alternative proposée par Vilím [57, 175].

Notes sur l'API On peut éventuellement nommer une ressource pour faciliter la lecture du modèle, par exemple avec `Solver.pretty()`.
Le nombre de tâches optionnelles est défini par la longueur du tableau `usages` (`usages.length` $\leq$ `tasks.length`). Si le paramètre `usages` est `null`, alors la ressource est régulière. En fait, les `usages.length` dernières tâches du tableau `tasks` sont optionnelles et les `tasks.length` − `usages.length` premières sont obligatoires. On peut bien sûr ignorer ce mécanisme en mélangeant les variables booléennes et constantes dans le tableau `usages`. Cependant, il est conseillé de respecter cette convention, car l'implémentation de l'interface `IRTask` est différente selon que la tâche soit obligatoire ou optionnelle.
Si l'utilisateur ne précise pas d'horizon de planification pour la ressource (`uppBound` = `null`) alors on le remplace par la variable `makespan` représentant la date d'achèvement de l'ordonnancement, en la créant automatiquement si nécessaire.
Les méthodes `cumulativeMax` considèrent que la consommation minimale est nulle (`consumption` = `null` ou 0). Les méthodes `cumulativeMin` considèrent que la capacité est infinie (`capacity` = `null`).

Exemple Un exemple d'utilisation de la contrainte `cumulative` est présenté en section B.2.

9.3.2 disjunctive

Signature `disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, IntegerVariable uppBound, String... options)`
`tasks` sont les tâches exécutées sur la ressource (recommandé $n \geq 3$).
`usages` indiquent si les tâches sont *effectives* sur la ressource (booléens).
`uppBound` est l'horizon de la ressource (obligatoire ≥ 0). .

Description La contrainte `disjunctive` est un cas particulier de la contrainte `cumulative` où les hauteurs et la capacité sont unitaires alors que la consommation minimale requise est nulle. La ressource n'exécute au plus une tâche à chaque instant, c'est-à-dire que les tâches ne se chevauchent pas.

Algorithme de filtrages Les règles de filtrage décrites en section 3.3.1 sont activées par les options. Nous avons implémenté les algorithmes de filtrage proposés par Vilím [43], Vilím *et al.* [44], Kuhnert [52], Vilím [150], Vilím *et al.* [176] qui changent selon le caractère régulier ou alternatif de la ressource.
Une règle atteint un point fixe local lorsqu'elle ne peut plus réaliser de déduction (ceci peut nécessiter plusieurs exécutions de la règle). La propagation des règles dans la contrainte doit atteindre un point fixe global qui est atteint lorsque toutes les règles ont atteint leur point fixe local.
Vilìm a proposé un algorithme de propagation où les règles atteignent successivement leur point fixe local jusqu'à ce que le point fixe global soit atteint dans la boucle principale. La boucle principale par défaut applique chaque règle une seule fois (une règle n'atteint pas forcément son point fixe local) jusqu'à ce que

le point fixe global soit atteint. Ce choix est motivé par des raisons spécifiques à notre implémentation, car il tente de réduire le nombre d'opérations de tri des arbres binaires équilibrés maintenus dans la contrainte.

options Les options sont des champs (`String`) de la classe `choco.Options`.
`NO_OPTION` filtrage par défaut : `C_DISJ_NFNL, C_DISJ_DP, C_DISJ_EF`.
`C_DISJ_OC` *overload checking* en $O(n \log n)$.
`C_DISJ_NFNL` *not first/not last* en $O(n \log n)$.
`C_DISJ_DP` *detectable precedence* en $O(n \log n)$.
`C_DISJ_EF` *edge finding* en $O(n \log n)$.
`C_DISJ_VF` applique l'algorithme de propagation des règles de Vilìm.

Notes sur l'API La sémantique des paramètres de l'API est identique à celle de la contrainte `cumulative`.
La méthode `Reformulation.disjunctive(TaskVariable[] clique, String... boolvarOptions)` décompose la contrainte en une clique de disjonctions grâce à `precedenceDisjoint` : $\texttt{clique}[i] \simeq \texttt{clique}[j],\ \forall\, i < j$.
La méthode `forbiddenIntervals(String name, TaskVariable[] tasks)` poste une contrainte globale appliquant la règle des intervalles interdits proposée par Guéret et Prins [120] (voir section 6.1.3).

9.3.3 useResources (en développement)

Signature `useResources(TaskVariable task, int k, Constraint... resources)`
`task` est la tâche à laquelle on alloue des ressources.
`k` est le nombre de ressources nécessaire à la tâche (obligatoire > 0).
`resources` sont les ressources à allouer.

Description La contrainte `useResources` représente l'allocation d'exactement ou au moins k ressources de l'ensemble `resources` à la tâche `task`. La représentation des tâches optionnelles dans les ressources alternatives `cumulative` et `disjunctive` par des variables booléennes `usages` permet de gérer l'allocation de ressources via des contraintes booléennes. Cependant, l'inconvénient des contraintes d'allocation booléennes est que les fenêtres de temps ne sont pas réduites avant une allocation effective. En effet, supposons qu'une tâche requiert deux ressources sur trois ressources disponibles mais qu'une seule ressource soit en fait disponible à sa date de début au plus tôt (les deux autres ressources sont occupées à cet instant), alors ni la contrainte booléenne d'allocation, ni les contraintes de ressource ne peuvent supprimer la date de début au plus tôt du domaine qui est pourtant inconsistante quelle que soit l'allocation.
Pour pallier cet inconvénient, nous introduisons une nouvelle contrainte *transversale*, c'est-à-dire que son scope contient aussi des contraintes et non uniquement des variables. Nous donnons plus de détails sur la définition de `useResources` et ses règles de filtrage en annexe D. Cette représentation offre une grande flexibilité par rapport aux contraintes multi-ressources, car on peut spécifier différentes demandes d'allocation sur un ensemble de ressources (une tâche requiert deux ressources et une autre requiert seulement une ressource parmi quatre) ou pour une tâche (une tâche nécessite deux machines, deux opérateurs et quatre travailleurs).
Notez que certains aspects de cette contrainte transversale sont difficiles à implémenter proprement dans choco. C'est pourquoi, pour l'instant, `useResources` est disponible dans `choco.ChocoContrib` mais pas incluse dans la livraison principale de choco.

Algorithme de filtrage Chaque tâche optionnelle d'une ressource alternative raisonne sur une fenêtre de temps *hypothétique*, c'est-à-dire en supposant qu'elle est effective sur la ressource (sans se soucier des autres contraintes). Cette fenêtre de temps est cohérente avec la fenêtre de temps principale de la tâche définie par les domaines de ses variables entières. Si la fenêtre de temps hypothétique devient vide, alors la tâche ne peut pas être exécutée sur la ressource. La contrainte `useResources` propage d'abord la contrainte booléenne d'allocation avant de réduire la fenêtre de temps principale en raisonnant sur les fenêtres de temps hypothétiques et l'allocation courante.

Notes sur l'API Remarquez que les contraintes `useResources` utilisent par défaut l'option `Options.C_POST_PONED`, car leur construction nécessite que toutes les contraintes de partage de ressource aient été préalablement créées dans le solveur.

9.3.4 precedence

Signature `precedence(TaskVariable t1, TaskVariable t2, int delta)`
`t1, t2` sont les tâches de la précédence.
`delta` est le temps d'attente entre les tâches.

Description La contrainte `precedence` représente une contrainte de précédence avec temps d'attente $\texttt{t1} \prec \texttt{t2}$ entre deux tâches non préemptives introduite en section 3.2 :

$$\texttt{t1.end} + \texttt{delta} \leq \texttt{t2.start} \qquad (\texttt{t1} \prec \texttt{t2})$$

Algorithme de filtrage On délègue au `Solver` le soin de poser la contrainte la mieux adaptée pour cette relation arithmétique. Ce choix dépend d'autres options globales du `Solver` comme `Options.E_DECOMP`. Cependant, les observations montrent qu'on applique généralement la consistance de bornes.

Exemple Un exemple d'utilisation de la contrainte `precedence` est présenté en section B.1.

9.3.5 precedenceDisjoint

Signature `precedenceDisjoint(TaskVariable t1, TaskVariable t2, IntegerVariable direction, int fs, int bs)`
`t1, t2` sont deux tâches en disjonction.
`dir` indique l'ordre relatif des deux tâches (booléen).
`fs, bs` sont les temps d'attente pour chaque arbitrage de la disjonction.

Description La contrainte `precedenceDisjoint` représente une contrainte réifiée de disjonction avec temps d'attente $\texttt{t1} \sim \texttt{t2}$ entre deux tâches non préemptives introduite au chapitre 7 :

$$\begin{aligned} \texttt{dir} = 1 \quad &\Rightarrow \quad \texttt{t1.end} + \texttt{fs} \leq \texttt{t2.start} \qquad (\texttt{t1} \prec \texttt{t2}) \\ \texttt{dir} = 0 \quad &\Rightarrow \quad \texttt{t2.end} + \texttt{bs} \leq \texttt{t1.start} \qquad (\texttt{t2} \prec \texttt{t1}) \end{aligned}$$

Algorithme de filtrage Dans un souci d'efficacité, le `Solver` utilise différentes implémentations selon la nature des durées des tâches (fixes ou non) et de l'existence de temps d'attente. Le filtrage applique les règles précédence interdite (PI), précédence obligatoire (PO) ou la consistance de borne sur un arbitrage. L'algorithme de filtrage réagit aux évènements en temps constant :

1. Si une modification d'une fenêtre de temps déclenche une des règles (PI) ou (PO), alors la variable `dir` est instanciée selon l'arbitrage déduit.
2. Après l'instanciation de la variable `dir`, on réduit les fenêtres de temps par consistance de borne jusqu'à ce que la précédence soit satisfaite.

Notez qu'une contradiction est déclenchée lorsqu'aucun séquencement n'est possible.

Exemple Un exemple d'utilisation de la contrainte `precedenceDisjoint` est présenté en section B.1.

9.3.6 precedenceReified

Signature `precedenceReified(TaskVariable t1, int delta, TaskVariable t2, IntegerVariable b)`
`t1, t2` sont les deux tâches de la précédence réifiée.
`delta` est temps d'attente entre les tâches.
`b` indique si la précédence ou son opposé au sens arithmétique est actif (booléen).

Description La contrainte `precedenceReified` la réification de la précédence $\texttt{t1} \prec \texttt{t2}$ en fonction de la valeur de `b`.

$$\begin{array}{lcll} \texttt{b} = 1 & \Rightarrow & \texttt{t1.end} + \texttt{delta} \leq \texttt{t2.start} & (\texttt{t1} \prec \texttt{t2}) \\ \texttt{b} = 0 & \Rightarrow & \texttt{t1.end} + \texttt{delta} > \texttt{t2.start} & (\texttt{t1} \not\prec \texttt{t2}) \end{array}$$

Algorithme de filtrage On applique une variante de l'algorithme de filtrage de `precedenceDisjoint` où la détection et la propagation du cas $\texttt{b} = 0$ est modifiée.

9.3.7 precedenceImplied

Signature `precedenceImplied(TaskVariable t1, int delta, TaskVariable t2, IntegerVariable b)`
`t1, t2` sont les deux tâches de la précédence potentielle.
`delta` est temps d'attente entre les tâches.
`b` indique si la précédence est active ou non (booléen).

Description La contrainte `precedenceImplied` représente l'activation ou non de la précédence $\texttt{t1} \prec \texttt{t2}$ en fonction de la valeur de `b`.

$$\begin{array}{lcll} \texttt{b} = 1 & \Rightarrow & \texttt{t1.end} + \texttt{delta} \leq \texttt{t2.start} & (\texttt{t1} \prec \texttt{t2}) \\ \texttt{b} = 0 & \Rightarrow & \texttt{TRUE} & \end{array}$$

Algorithme de filtrage On applique une relaxation de l'algorithme de filtrage de `precedenceDisjoint`.

9.3.8 disjoint (décomposition)

Signature `disjoint(TaskVariable[] tasks1, TaskVariable[] tasks2)`
`tasks1, tasks2` sont deux ensembles de tâches en disjonction.

Description La contrainte `disjoint` impose que les paires de tâches composées d'une tâche de chaque ensemble soient en disjonction : $(\texttt{tasks1}[i] \simeq \texttt{tasks2}[j])$, c'est-à-dire qu'elles ne se chevauchent pas dans le temps. La méthode renvoie un objet de la classe `Constraint[]` qui modélise une décomposition de la contrainte globale `disjoint` en contraintes `precedenceDisjoint`. En effet, il n'existe aucune implémentation de cette contrainte globale, mais la cumulative colorée (indisponible dans `choco`) permet de reformuler efficacement cette contrainte en affectant une couleur différente à chaque ensemble de tâches et en imposant une capacité unitaire à la ressource. La cumulative colorée impose que le nombre de couleurs distinctes appartenant aux tâches s'exécutant à l'instant t reste inférieure à la capacité de la ressource.

9.3.9 pack

Signature `pack(SetVariable[] itemSets, IntegerVariable[] loads, IntegerVariable[] bins, Integer-ConstantVariable[] sizes,IntegerVariable nbNonEmpty, String... options)`
`items` sont les articles à ranger dans les conteneurs.
`loads` sont les charges des conteneurs.
`bins` sont les conteneurs dans lesquels sont rangés les articles (≥ 0).
`sizes` sont les tailles (longueurs) des articles (constantes).
`nbNE` est le nombre de conteneurs non vide du rangement.

Description La contrainte `pack(items, loads, bins, sizes, nbNE)` impose qu'un ensemble d'articles soient rangés dans des conteneurs de manière à ce que la somme des tailles des articles dans un conteneur soit cohérente avec sa charge (9.5). La contrainte assure le respect des contraintes de liaison

entre les variables représentant le rangement (9.6), mais aussi la cohérence entre le nombre de conteneurs non vides et l'affectation des articles 9.7.

$$\texttt{loads}[b] = \sum_{i \in \texttt{items}[b]} \texttt{sizes}[i] \quad (\forall \text{ conteneur } b) \tag{9.5}$$

$$\texttt{bins}[i] = b \iff i \in \texttt{items}[b] \quad (\forall \text{ article } i)(\forall \text{ conteneur } b) \tag{9.6}$$

$$\text{card}\{b \mid \texttt{items}[b] \neq \emptyset\} = \texttt{nbNE} \tag{9.7}$$

Les deux restrictions principales sont le tri obligatoire des articles par taille décroissante et les tailles constantes des articles. En effet, l'implémentation n'est pas robuste à un tri des variables `bins` et `sizes` puisque les ensembles `items` référencent les articles par leur index dans ces tableaux. Un tri intempestif peut entraîner des effets de bords sur d'autres contraintes portant sur les variables ensemblistes. À terme, l'intégration de tailles variables ne devrait pas poser de problèmes mais elle peut compliquer voire empêcher l'application de certaines règles de filtrage, notamment `C_PACK_AR`. Cette contrainte générique permet de modéliser facilement des conteneurs de tailles hétérogènes ou des incompatibilités entre article/article ou article/conteneur.

Algorithme de filtrage Notre implémentation s'inspire librement de Shaw [70] puisque l'ajout des variables `items` et `nbNE` modifie la signature de la contrainte. Tout d'abord, la contrainte applique la propagation associée au *Typical model*, c'est-à-dire les contraintes (8.7) and (8.8) de la formulation mathématique présentée en section 8.4.3 page 88. Par ailleurs, Le filtrage applique des règles inspirées de raisonnements sur les problèmes de sac à dos (*additional propagation rules*) ainsi qu'une borne inférieure dynamique sur le nombre minimal de conteneurs non vides (*using a lower bound*).
Les raisonnements de type sac à dos consistent à traiter le problème plus simple du rangement des articles dans un seul conteneur : existe-t'il un rangement d'un sous-ensemble d'articles dans un conteneur b pour lequel la charge du conteneur appartient à un intervalle donné ? S'il n'existe aucun rangement d'un sous-ensemble d'articles dans le conteneur b pour lequel la charge du conteneur appartient au domaine de $\texttt{loads}[b]$, alors on peut déclencher un échec. Par ailleurs, si un article apparaît dans tous ces rangements, alors cet article est immédiatement rangé dans le conteneur. Au contraire, si un un article n'apparaît dans aucun rangement, alors cet article est éliminé des candidats pour ce conteneur. Ainsi, si aucun rangement réalisable dans un conteneur n'atteint une charge donnée, alors cette charge n'est pas réalisable. Shaw [70] a proposé des algorithmes efficaces ne dépendant pas des tailles et capacités, mais qui n'atteignent pas la consistance d'arc généralisée sur le problème *subset sum* à l'image de ceux proposés par Trick [73]. Ensuite, il adapte les bornes inférieures classiques pour ce problème pour les affectations partielles. Le principe consiste à transformer une affectation partielle en une nouvelle instance du problème de *bin packing*, puis d'appliquer une borne inférieure sur cette nouvelle instance.
Nous considérons aussi deux règles d'élimination dynamique des symétries qui sont fréquemment utilisées pour le problème de *bin packing*. D'autres règles d'élimination statique des symétries sont proposées dans la classe `PackModel` décrite ci-dessous.

options Les options sont des champs (`String`) de la classe `choco.Options`.
`NO_OPTION` filtrage par défaut : propagation du *Typical model* [70].
`C_PACK_AR` voir section *Additional propagation rules* [70].
`C_PACK_DLB` voir section *Using a lower bound* [70]. Nous remplaçons la borne inférieure L^{1D}_{MV} par L^{1D}_{CCM} [82] qui est dominante.
`C_PACK_FB` élimination des symétries : on range immédiatement un article dans un conteneur si l'article remplit complètement le conteneur.
`C_PACK_LBE` élimination des symétries : les conteneurs vides sont à la fin.

Notes sur l'API Un objet de la classe `PackModel` facilite la modélisation par le biais de ses nombreux constructeurs et de méthodes postant des contraintes d'élimination des symétries dont certaines sont décrites dans le tableau 9.4. Nous attirons l'attention du lecteur sur le fait que la validité de ces dernières dépend du problème traité et qu'on ne peut jamais les appliquer toutes simultanément.

Méthode *PackModel*	Description
`PackModel(int[], int , int)`	constructeur basique pour le problème de *bin packing*.
`packLargeItems()`	range les k plus grands articles dans les k premiers conteneurs. La valeur de k dépend des tailles et de de la capacité.
`allDiffLargeItems()`	contrainte redondante `allDifferent` sur les k plus grands articles.
`orderEqualSizedItems(int)`	ordonne les articles de taille identique dans les conteneurs.
`decreasingLoads(int)`	ordonne les conteneurs par charge décroissante.
`decreasingCardinalities(int)`	ordonne les conteneurs par nombre d'articles décroissant.

TABLE 9.4 – Principales méthodes de la classe `PackModel`.

Exemple Un exemple d'utilisation de la contrainte `pack` est présenté en section B.3.

9.3.10 Reformulation de contraintes temporelles

Le tableau 9.5 récapitule les méthodes de la classe `Choco` qui aident à la création et à la lisibilité de modèles utilisant des contraintes temporelles. Ces méthodes proposent une interface de reformulation de contraintes temporelles vers des contraintes arithmétiques. Le paramètre `delta` n'est pas obligatoire, c'est-à-dire il existe toujours une méthode du même nom dont la signature ne contient pas ce paramètre (la valeur est alors considérée nulle).

Méthode Choco.*	Reformulation
Contraintes sur les fenêtres de temps	
`endsAt(TaskVariable t, int time)`	$\texttt{t.end} = \texttt{time}$
`endsAfter(TaskVariable t, int ect)`	$\texttt{t.end} \geq \texttt{ect}$
`endsBefore(TaskVariable t, int lct)`	$\texttt{t.end} \leq \texttt{lct}$
`endsBetween(TaskVariable t, int ect, int lct)`	$\texttt{ect} \leq \texttt{t.end} \leq \texttt{lct}$
`startsAt(TaskVariable t, int time)`	$\texttt{t.start} = \texttt{time}$
`startsAfter(TaskVariable t, int est)`	$\texttt{t.start} \geq \texttt{est}$
`startsBefore(TaskVariable t, int lst)`	$\texttt{t.start} \leq \texttt{lst}$
`startsBetween(TaskVariable t, int est, int lst)`	$\texttt{est} \leq \texttt{t.start} \leq \texttt{lst}$
Contraintes d'enchaînement	
`startsBeforeBegin(TaskVariable t1, TaskVariable t2, int delta)`	$\texttt{t1.start} + \texttt{delta} \leq \texttt{t2.start}$
`startsAfterBegin(TaskVariable t1, TaskVariable t2, int delta)`	$\texttt{t1.start} \geq \texttt{t2.start} + \texttt{delta}$
`startsBeforeEnd(TaskVariable t1, TaskVariable t2, int delta)`	$\texttt{t1.start} + \texttt{delta} \leq \texttt{t2.end}$
`endsAfterBegin(TaskVariable t1, TaskVariable t2, int delta)`	$\texttt{t1.end} \geq \texttt{t2.start} + \texttt{delta}$
`endsBeforeEnd(TaskVariable t1, TaskVariable t2, int delta)`	$\texttt{t1.end} + \texttt{delta} \leq \texttt{t2.end}$
`endsAfterEnd(TaskVariable t1, TaskVariable t2, int delta)`	$\texttt{t1.end} \geq \texttt{t2.end} + \texttt{delta}$
`endsBeforeBegin(TaskVariable t1, TaskVariable t2, int delta)`	`precedence(t1, t2, delta)`
`startsAfterEnd(TaskVariable t1, TaskVariable t2, int delta)`	`precedence(t2, t1, delta)`

TABLE 9.5 – Reformulation des contraintes temporelles.

9.4 Construction du modèle disjonctif

Nous décrivons une méthode générale pour la construction d'un modèle disjonctif généralisé $\mathcal{G}$ défini en section 3.4 à partir d'un modèle de base défini par l'utilisateur. Elle gère des contraintes hétérogènes provenant de sources différentes et potentiellement redondantes. Son intérêt est plus restreint dans le

cadre de problèmes académiques très structurés, car on peut alors appliquer les règles de reformulation et de déduction de manière *ad hoc*. Cependant, cette méthode a permis un gain de temps significatif lors de l'implémentation des modèles *Light* et *Heavy* du chapitre 7.
La construction du graphe disjonctif est activée par la valeur de la propriété booléenne `DISJUNCTIVE_MODEL_DETECTION` de la classe `PreProcessConfiguration`. La construction modulaire (on peut supprimer certaines étapes) du graphe disjonctif généralisé dépend d'autres propriétés de cette classe. La procédure assure l'unicité d'une contrainte temporelle entre deux tâches.

On distingue les opérations sur le graphe disjonctif de celles sur le modèle. Les opérations sur le graphe disjonctif ont la syntaxe suivante : `opération` élément(s) $\multimap$ ensemble(s). Les opérations sur le modèle (leur nom commence avec une majuscule) ont la syntaxe suivante : `Opération` élément(s).

9.4.1 Traitement des contraintes temporelles

La procédure buildDisjModT détecte les précédences issues des fenêtres, puis construit le graphe disjonctif en se basant sur les contraintes `precedence` et `precedenceDisjoint` du modèle. Après l'initialisation du graphe $\mathcal{G}$, on calcule un ensemble d'arcs fictifs $\mathcal{P}_0$ qui représente la relation d'ordre entre les fenêtres de temps des tâches : $(i,j) \in \mathcal{P}_0 \Leftrightarrow est_j - lct_i >= 0$. Cette étape modifie la valuation $\mathcal{L}$ sans modifier les arcs de $\mathcal{P}$. Ces arcs fictifs permettent d'éliminer les précédences satisfaites et les disjonctions arbitrées par les fenêtres de temps des tâches.

On parcourt d'abord les contraintes de précédence du modèle. Une contrainte `precedence` est supprimée du modèle lorsqu'elle est dominée par la valuation courante de $\mathcal{G}$. Dans le cas contraire, si l'arc est fictif, alors on ajoute simplement la contrainte dans $\mathcal{G}$. Si l'arc est déjà associé à une contrainte, on fusionne les contraintes pour mettre à jour la valuation et la contrainte dans $\mathcal{G}$. À la fin de cette boucle, le graphe de précédence est construit et nous calculons sa fermeture transitive qui va être utile dans le traitement des disjonctions. Pendant ce calcul, une sous-fonction lève une contradiction lorsqu'un cycle est détecté dans le graphe de précédence. Par convention, les arcs transitifs ont un temps d'attente nul.

On parcourt ensuite les contraintes de disjonction du modèle. Si T_j appartient à la fermeture transitive de T_i, la disjonction est déjà arbitrée. Si la précédence associée à cet arbitrage a un temps d'attente supérieur à celui existant dans $\mathcal{G}$, alors on distingue deux cas selon l'existence d'un arc dans $\mathcal{G}$. Si l'arc est fictif, on ajoute cette nouvelle précédence dans le modèle et dans $\mathcal{G}$. Sinon, on fusionne la nouvelle précédence avec celle de $\mathcal{G}$. La disjonction est alors supprimée du modèle et sa variable booléenne est instanciée en fonction de l'arbitrage déduit. L'élimination d'une disjonction transforme $\mathcal{G}$ sans modifier sa fermeture transitive.
Lorsque la disjonction n'est pas encore arbitrée, on distingue deux cas selon l'existence d'une arête dans $\mathcal{G}$. Si une arête existe, on fusionne les deux contraintes de disjonction dans $\mathcal{G}$, puis on remplace la variable booléenne b_{ij} associée à la disjonction par celle de $\mathcal{G}$. Dans le cas contraire, on ajoute la disjonction à $\mathcal{G}$.

Finalement, on calcule une approximation de la réduction transitive du graphe de précédence où une sous-fonction élimine les arcs de valuation nulle sont éliminés. En effet, l'élimination d'un arc de valuation strictement positive requiert l'évaluation des longueurs des chemins entre les deux tâches. De notre point de vue, un tel algorithme relève de la propagation puisqu'il entraîne des modifications des fenêtres de temps des tâches qui permet de déduire de nouvelles précédences jusqu'à atteindre un point fixe. De plus, ce raisonnement est renforcé par la propagation des autres contraintes du modèle. Malgré quelques considérations numériques simples, notre méthode est structurelle puisqu'elle repose sur l'analyse du réseau de contraintes et non des domaines. Actuellement, le calcul de la fermeture et de la réduction transitive n'est pas encore implémenté, car il n'était pas nécessaire pour les problèmes d'atelier.

9.4.2 Traitement des contraintes de partage de ressource

La procédure buildDisjModR analyse l'occupation des ressources (`cumulative` et `disjunctive`) pour détecter de nouvelles disjonctions. Pour chaque ressource, on détermine si un couple de tâches obligatoires (T_i, T_j) définit une nouvelle disjonction. Pour ce faire, on vérifie qu'il n'existe aucun chemin du graphe de précédence ou disjonction entre ces deux tâches et que leur exécution simultanée sur la ressource provoque un dépassement de sa capacité. Dans ce cas, on ajoute une contrainte de disjonction sans temps d'attente entre les deux tâches dans le modèle et dans $\mathcal{G}$.

Procédure buildDisjModT : détection des précédences.

initialiser $\mathcal{G} = (\mathcal{T}, \mathcal{P}, \mathcal{D}, \mathcal{A})$; // $\mathcal{T} \subset \mathcal{X}$, $\mathcal{P} = \mathcal{D} = \emptyset$, $l(i,j) = -\infty$
si estActif(DMD_USE_TIME_WINDOWS) alors calculer $\mathcal{P}_0 \multimap \mathcal{L}$;
pour chaque $T_i \prec T_j$ **faire**
  /* contraintes precedence */
  si $d_{ij} > l(i,j)$ **alors**
    si $(T_i, T_j) \in \mathcal{P}$ **alors**
      fusionner $T_i \prec T_j \multimap \mathcal{P}, \mathcal{L}$;
      Retracter $T_i \prec T_j$;
    sinon ajouter $T_i \prec T_j \multimap \mathcal{P}, \mathcal{L}$;
  sinon Retracter $T_i \prec T_j$;
calculer la fermeture transitive ($\rightsquigarrow$) du graphe partiel engendré par les arcs $\mathcal{P}_0 \cup \mathcal{P}$;
  sous-fonction lever une contradiction lorsqu'un cycle est détecté;
  pour chaque $T_i \sim T_j$ **faire**
    /* contraintes precedenceDisjoint */
    si $T_i \rightsquigarrow T_j$ **alors**
      /* $T_i \prec T_j$ est la contrainte dérivée de $T_i \sim T_j$ */
      si $d_{ij} > l(i,j)$ **alors**
        si $(T_i, T_j) \in \mathcal{P}$ **alors** fusionner $T_i \prec T_j \multimap \mathcal{P}, \mathcal{L}$;
        sinon
          Poster $T_i \prec T_j$;
          ajouter $T_i \prec T_j \multimap \mathcal{P}, \mathcal{L}$;
      Retracter $T_i \sim T_j$;
      Poster $b_{ij} = 1$;
    sinon si $T_j \rightsquigarrow T_i$ **alors** ... ; // condition réciproque
    si $(T_i, T_j) \in \mathcal{D}$ **alors**
      fusionner $T_i \sim T_j \multimap \mathcal{D}, \mathcal{L}$;
      Remplacer b_{ij} par la variable booléenne associée à (T_i, T_j) ;
    sinon ajouter $T_i \sim T_j \multimap \mathcal{D}, \mathcal{L}$;
  calculer la réduction transitive du graphe partiel engendré par les arcs $\mathcal{P}_0 \cup \mathcal{P}$;
    sous-fonction supprimer uniquement les arcs de valuation nulle (sans temps d'attente);

Procédure buildDisjModR : traitement des ressources.

pour chaque $r \in \mathcal{R}$ **faire**
  pour chaque $(T_i, T_j) \in r$ **such that** $i < j$ **et** $u_i(r) = 1$ **et** $u_j(r) = 1$ **faire**
    /* T_i et T_j sont des tâches obligatoires */
    si $T_i \not\rightsquigarrow T_j$ **et** $T_j \not\rightsquigarrow T_i$ **et** $(T_i, T_j) \notin \mathcal{D}$ **et** $\min(h_i^r) + \min(h_i^r) > \max(C_r)$ **alors**
      Poster $T_i \simeq T_j$; // $d_{ij} = d_{ji} = 0$
      ajouter $T_i \simeq T_j \multimap \mathcal{D}, \mathcal{L}$;
  si $C_r = 1$ **et** estActif(DMD_REMOVE_DISJUNCTIVE) **alors** Retracter r;
  si $C_r > 1$ **et** estActif(DISJUNCTIVE_FROM_CUMULATIVE_DETECTION) **alors**
    $T \leftarrow \{T_i \in r \mid 2 \times \min(h_i^r) > \max(C_r)\}$;
    $U \leftarrow \{u_i(r) \mid T_i \in T\}$;
    si $|T| > 3$ **alors** Poster disjunctive (T, U) ;

Après cette étape, une option peut déclencher la suppression de la contrainte `disjunctive` puisqu'elle a été décomposée en contraintes `precedenceDisjoint`. Cette suppression entraîne une réduction du filtrage puisque les règles *not first/not last*, *detectable precedence* et *edge finding* ne sont alors plus appliquées.
Au contraire, une autre option déclenche le renforcement de la propagation de la contrainte `cumulative`. En effet, on peut poster une contrainte `disjunctive` sur la plus grande une clique disjonctive de la contrainte `cumulative`. Notez que la clique peut contenir des tâches obligatoires et optionnelles.

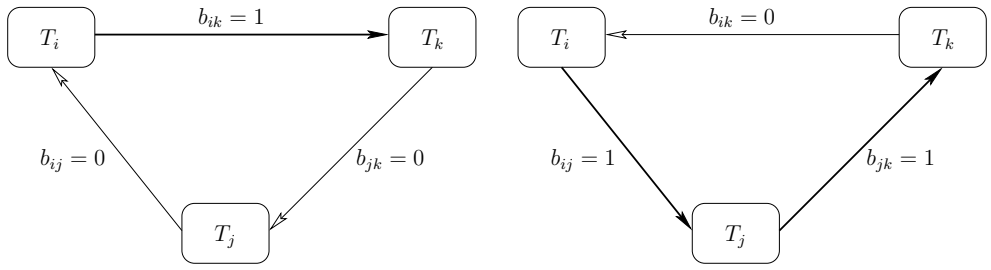
9.4.3 Déduction de contraintes « coupe-cycle »

Supposons que notre modèle contienne une clique disjonctive contenant trois tâches T_i, T_j et T_k. Si une affectation partielle contient les arbitrages $T_i \prec T_j$ et $T_j \prec T_k$, alors l'arbitrage $T_i \prec T_k$ peut être inféré par transitivité (inégalité triangulaire), car l'existence d'un cycle engendre une inconsistance. Les contraintes `precedenceDisjoint` sont propagées individuellement et les arbitrages par transitivité ne sont pas détectés. Cette remarque est valable pour des cliques de taille quelconque.
Dans les problèmes de tournées de véhicules, une problématique similaire est résolue en posant un nombre exponentiel de contraintes d'élimination des sous-tours [177]. La procédure postCoupeCycle pose un nombre quadratique de clauses booléennes similaires aux contraintes d'élimination des sous-tours de longueur 3. Dans le cas général, ces contraintes ne détectent pas forcément tous les arbitrages transitifs, par exemple, lorsqu'il existe un cycle de longueur 4 constitué de tâches ne formant pas une clique disjonctive. Par contre, ces contraintes sont suffisantes pour la décomposition des contraintes `disjunctive`. Ces contraintes peuvent réduire le nombre d'arbitrages visités pendant la résolution.

Notons b_{ij} la variable booléenne réifiant la contrainte de disjonction $T_i \sim T_j$. Pour chaque couple d'arêtes adjacentes d'extrémités T_i et T_k, on simplifie la contrainte d'élimination des sous-tours en fonction de l'existence d'un arbitrage d'une des deux disjonctions. La figure 9.1 illustre le cycle du graphe disjonctif éliminé par chaque clause ou sa simplification. Par souci de clarté, nous ignorons les éventuelles opérations de substitution sur les variables booléennes ($b_{ij} = \neg b_{ji}$) relatives à l'« orientation » des contraintes du modèle ($T_i \sim T_j$ ou $T_j \sim T_i$).

Procédure postCoupeCycle : ajout des contraintes d'élimination des sous-tours de longueur 3.

pour chaque $T_i \sim T_j,\ T_j \sim T_k \in \mathcal{D}$ **faire**
  si $T_i \rightsquigarrow T_k$ **alors**
    `Poster clause` $(b_{ij} \vee b_{jk})$;
  sinon si $T_k \rightsquigarrow T_i$ **alors**
    `Poster clause` $(\neg b_{ij} \vee \neg b_{jk})$;
  sinon si $(i,k) \in \mathcal{D}$ **alors**
    `Poster clause` $(\neg b_{ik} \vee b_{ij} \vee b_{jk})$;
    `Poster clause` $(b_{ik} \vee \neg b_{ij} \vee \neg b_{jk})$;



(a) $\neg b_{ik} \vee b_{ij} \vee b_{jk}$ (b) $b_{ik} \vee \neg b_{ij} \vee \neg b_{jk}$

FIGURE 9.1 – Élimination des arbitrages créant un cycle de longueur 3 dans le graphe disjonctif.

9.5 CSP Solver Competition 2009

Le but de la *CSP solver competition* (CSC) est d'améliorer notre connaissance sur l'efficacité des différents mécanismes disponibles dans les solveurs de contraintes tels que les algorithmes de recherche, stratégies de branchement et les algorithmes de filtrage. On considère la résolution de CSP, Max-CSP (on autorise la violation des contraintes et la qualité de la solution est estimée en fonction du nombre de contraintes satisfaites) et Weighted-CSP (un poids est alloué à la violation d'une contrainte et on cherche une solution dont le poids des violations est minimal). Notez qu'on ne considère pas les problèmes d'optimisation. On considère des contraintes binaires ou non définies en extension ou en intension ainsi que certaines contraintes globales définies sur des variables entières. Un solveur peut participer à la compétition même s'il ne peut résoudre toutes les catégories de problèmes.
Nous présentons dans cette section les résultats de la CSC'2009 sur les problèmes d'atelier (catégorie 2-ARY-INT) et d'ordonnancement cumulatif (catégorie Alldiff+Cumul+Elt+WSum). Neuf équipes ont participé à cette édition et proposé quatorze solveurs tous basés sur des logiciels libres (aucun solveur commercial ne participe à cette compétition).
La résolution d'une instance par un solveur est interrompue après 1800 secondes. Le *virtual best solver* (VBS) est une construction théorique qui renvoie la meilleure réponse fournie par un des solveurs participants. On peut le voir comme un méta-solveur qui utilise un oracle parfait déterminant le meilleur solveur sur une instance donnée ou comme un solveur exécutant en parallèle tous les solveurs participants.

Les deux solveurs basés sur `choco` participent à toutes les catégories de problèmes. Ils utilisent notamment le module décrit dans l'annexe A. Un processus de configuration automatique est nécessaire pour obtenir des performances raisonnables sur les différentes catégories. Tout d'abord, plusieurs méthodes structurelles d'analyse et de reformulation sont appliquées sur le modèle encodé dans un fichier d'instance. Ensuite, la configuration de l'algorithme de recherche est basée sur le résultat d'une étape de *shaving* pendant la propagation initiale. Cette étape est éventuellement interrompue brutalement par le déclenchement d'une limite de temps. L'intérêt du *shaving* est renforcé par la présence de nombreuses instances insatisfiables dans toutes les catégories, car celui-ci détecter une inconsistance avant la recherche. Après cette étape, on configure la stratégie de branchement et celle de redémarrage en exploitant les informations sur les domaines et le réseau de contraintes. On sélectionne une des heuristiques de sélection de variable (voir section 2.3) : *lex*, *random*, *dom/deg*, *dom/wdeg*, *impact*. L'heuristique de sélection de valeur est aléatoire. On lance ensuite un algorithme de recherche en profondeur d'abord avec retour arrière. À notre connaissance, il n'y a pas de différence majeure entre les deux solveurs sur les catégories discutées présentement.

9.5.1 Catégorie 2-ARY-INT

Nous nous intéressons d'abord à la résolution des instances d'open-shop et de job-shop (catégorie 2-ARY-INT). Ces instances sont dérivées des jeux d'instances pour l'optimisation de Taillard et Guéret-Prins par une transformation vers des problèmes de satisfaction en fixant un horizon de planification (satisfiable ou non). Les durées des opérations subissent éventuellement un processus de perturbation ou de normalisation. Le modèle encodé est basé sur des contraintes de disjonction (non réifiées) entre paires de tâches.
Nous appliquons en temps polynomial plusieurs méthodes de reformulation pour construire un modèle similaire au modèle *Heavy* (voir section 7.1) :

1. Identification des tâches appartenant aux disjonctions du modèle encodé.
2. Réification des contraintes de disjonctions par transformation en contraintes `precedenceDisjoint`.
3. Ajout de contraintes redondantes `disjunctive` par identification des cliques de disjonction.

Par contre, nous n'appliquons pas les stratégies de recherche proposées aux chapitres 6 et 7. Contrairement à la reformulation, elles nous apparaissent très dépendantes des problèmes d'atelier et ne correspondent donc pas à l'esprit du concours.

Nous récapitulons le nombre de réponses données par chaque solveur après la résolution de toutes les instances d'open-shop en figure 9.2 ou de job-shop en figure 9.3 : satisfiables (SAT), insatisfiables (UNSAT) ou non résolues (UNKNOWN). Notez que les solveurs peuvent donner le même nombre de réponses sans toutefois que ce soit sur les mêmes instances. `choco` résout le plus grand nombre d'instances d'open-shop et ses résultats sont très proches de ceux du VBS pour les instances insatisfiables. Les

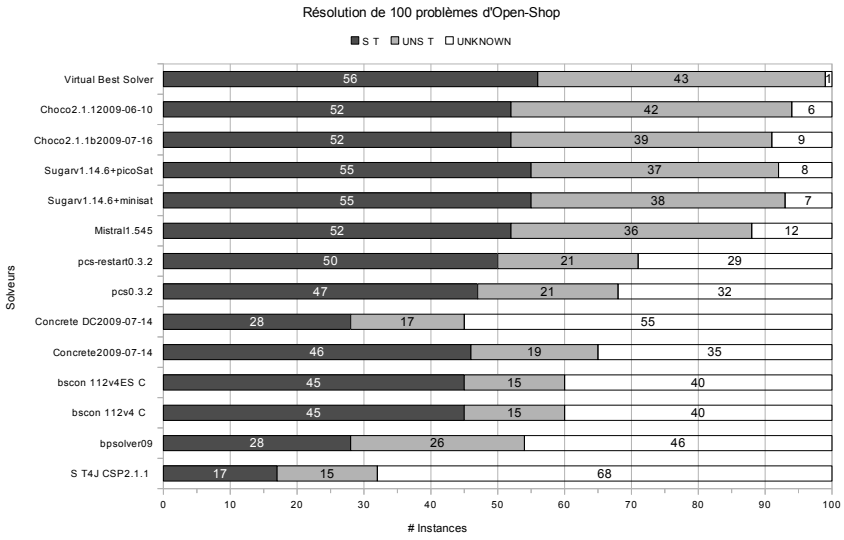


FIGURE 9.2 – Résolution de 100 problèmes d'open-shop (CSC'2009).

résultats de trois autres solveurs (`sugar +minisat`, `sugar +picosat`, `mistral`) rivalisent avec ceux de `choco`. Les solveurs `sugar` trouvent plus de solutions (SAT), mais moins d'inconsistances (UNSAT) que `choco`. Les résultats de `choco` sur les instances de job-shop restent parmi les meilleurs. Cependant, les

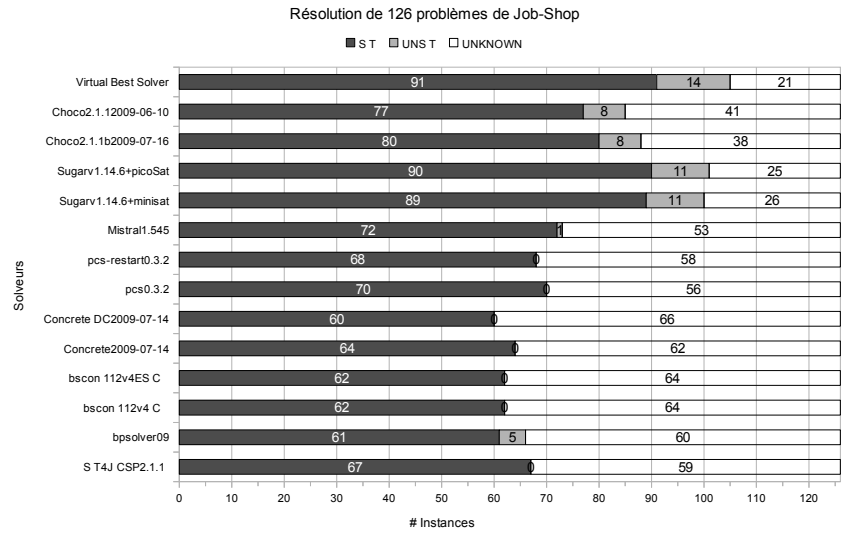


FIGURE 9.3 – Résolution de 126 problèmes de job-shop (CSC'2009)

solveurs `sugar` s'approchent plus du VBS que ce soit pour les instances satisfiables ou insatisfiables. Par contre, les résultats de `mistral` se dégradent légèrement. D'une manière générale, les différences entre les solveurs s'accentuent, notamment sur les instances insatisfiables détectées uniquement par quatre solveurs (`choco`, `sugar`, `mistral`, `bpsolver09`). En considérant toutes les instances, `choco` et `sugar` donnent respectivement le plus grand nombre de réponses insatisfiables et satisfiables.
les graphes de la figure 9.4 représentent le nombre d'instances qu'un solveur est capable de résoudre dans une limite de temps. Les instances d'open-shop et de job-shop sont groupées selon les réponses définitives de chaque solveur : SAT+UNSAT (en haut), SAT (en bas à gauche), UNSAT (en bas à

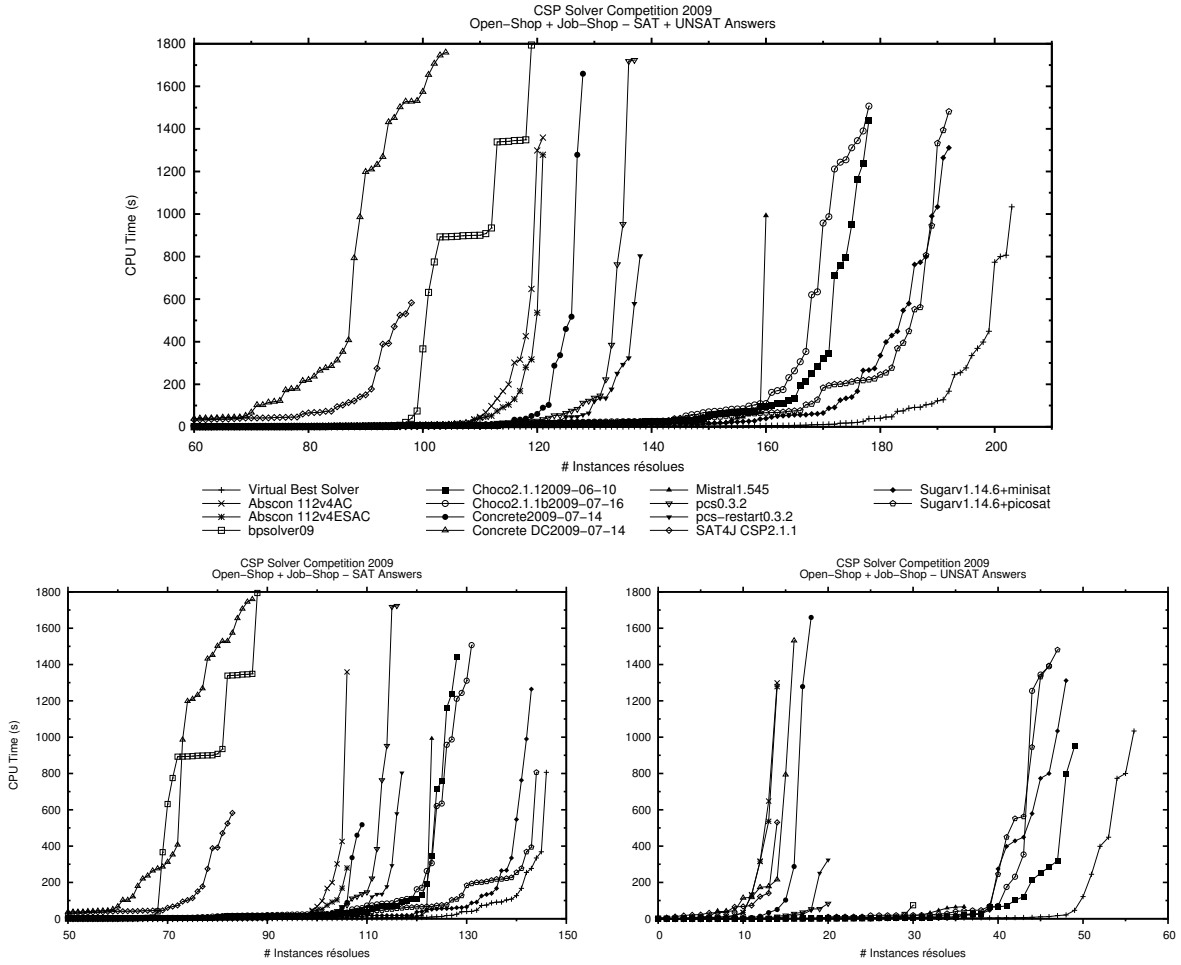


FIGURE 9.4 – Nombre d'instances résolues dans une limite de temps (2-ARY-INT).

droite). La coordonnée x est le nombre d'instances résolues par un solveur et la coordonnée y est la limite de temps en secondes pour la résolution de chaque instance. La transition de phase d'un solveur est caractérisée par sa position et sa forme. Cette transition a lieu quand les temps de résolution augmentent rapidement. Lorsque la transition a presque la forme d'un escalier, la réponse définitive du solveur est presque immédiate et celui-ci n'est pas capable de profiter du temps alloué supplémentaire. Lorsque la transition a une forme étendue, les réponses définitives du solveur sont dispersées sur une partie de l'intervalle de temps alloué.

Pour les réponses SAT+UNSAT (en haut), la courbe du VBS (à droite) est éloignée des autres ce qui laisse supposer que les temps de résolution d'une même instance varient beaucoup d'un solveur à l'autre. En d'autres termes, pour chaque instance résolue, il existe souvent un solveur pour lequel cette instance est résolue rapidement. Ensuite, ce graphe confirme que les solveurs les plus efficaces sont aussi les plus rapides (`choco`, `sugar`, `mistral`). La transition de phase de `mistral` a une forme en escalier contrairement à celles de `choco` et `sugar` qui sont plus étendues. D'ailleurs, la transition de phase plus étendue de `sugar` capture plus de réponses définitives. Les réponses SAT (graphe en bas à gauche) sont plus défavorables à `choco` puisque la courbe de `mistral` croise celles de `choco` et que les courbes de `sugar` se rapprochent de celle du VBS. La situation change pour les réponses UNSAT (graphe en bas à droite) où `choco` est le plus efficace et le plus rapide. Le solveur `mistral` reste rapide mais sa transition de phase est plus précoce.
Les solveurs `sugar` appliquent une technique d'encodage d'un CSP vers un problème de satisfiabilité booléenne similaire à celle de SAT-Ta discutée au chapitre 6. Dans le cadre de la CSC'2009, la comparaison

des résultats de choco et sugar sur les instances satisfiables confirment les conclusions des chapitres 6 et 7 à propos de l'importance des autres composantes de notre approche (solution initiale, stratégie de branchement). Ce phénomène est encore plus flagrant pour le solveur mistral dont les performances sont significativement dégradées et qui devient moins rapide que choco. À notre connaissance, choco est le seul solveur qui utilise une contrainte globale disjunctive. Ainsi, le succès de choco sur les instances insatisfiables montre que ces inférences supplémentaires permettent effectivement de prouver l'inconsistance du modèle plus rapidement, notamment pendant l'étape de *shaving*. En conclusion, nous soulignons que les résultats obtenus pour résolution des problèmes d'atelier figurent parmi les meilleures obtenues par choco au cours de cette compétition.

9.5.2 Catégorie Alldiff+Cumul+Elt+WSum

Nous nous intéressons maintenant à la résolution des instances pour les problèmes d'ordonnancement cumulatif (catégorie Alldiff+Cumul+Elt+WSum) comme le *resource-constrained project scheduling problem* [159]. Nous évaluons ainsi l'implémentation de la contrainte cumulative. Les graphes de la figure 9.5 représentent le nombre d'instances qu'un solveur est capable de résoudre dans une limite de temps pour les réponses définitives : SAT (à gauche), UNSAT (à droite). Quatre solveurs seulement participent à cette catégorie : choco, sugar, mistral et bpsolver09. Pour les réponses SAT, choco ne se place que troisième devant bpsolver09 et légèrement derrière mistral alors que sugar obtient sans conteste les meilleurs résultats. Pour les réponses UNSAT, choco obtient les meilleurs résultats en dominant légèrement sugar en terme de nombre réponses grâce à une transition de phase plus étendue. Les performances de mistral et surtout de bpsolver09 se situent un peu en retrait pour les réponses UNSAT.

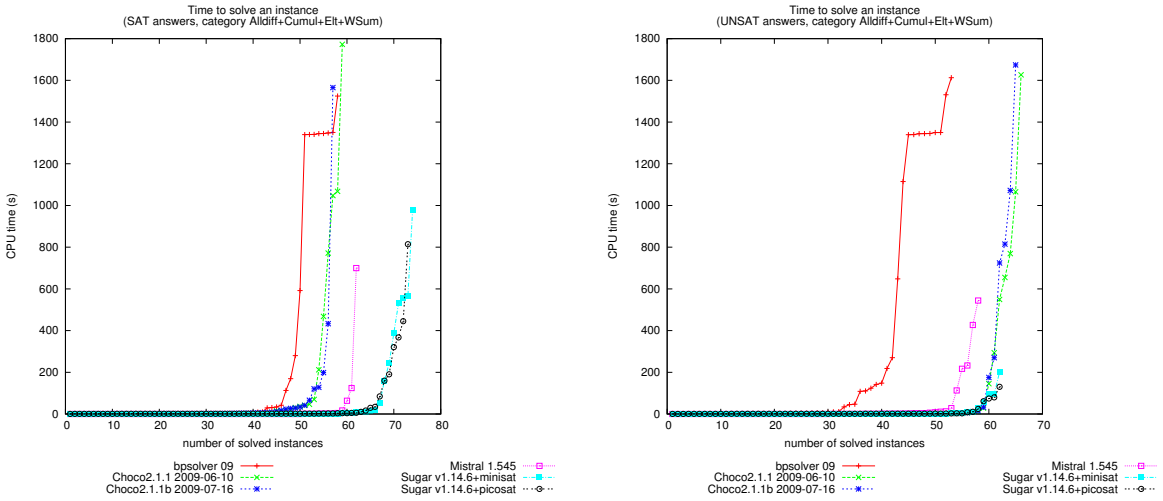


FIGURE 9.5 – Nombre d'instances résolues dans une limite de temps (Alldiff+Cumul+Elt+WSum).

9.6 Conclusion

Nous avons présenté nos principales contributions au solveur de contraintes choco. Elles concernent principalement l'ordonnancement, le placement et les redémarrages. Ce travail a aussi engendré une amélioration de certains mécanismes internes du solveur. Les résultats de la CSC'2009 confirment les résultats obtenus dans cette thèse quant aux bonnes performances de ces modules. Ces modules sont suffisamment stables pour que d'autres contributeurs puissent participer à leur développement. Ainsi, l'*edge finding* pour la contrainte disjunctive alternative a déjà été implémenté [6]. Une contrainte *soft* cumulative [178] et l'*edge finding* cumulatif sont actuellement en développement. Ces modules ont aussi permis de résoudre des problèmes de reconfiguration [179] et de placement [180].

Chapitre 10

Conclusion

Nous avons étudié au cours de cette thèse la résolution de problèmes d'ordonnancement par la programmation par contraintes. Plutôt que d'améliorer des techniques ou algorithmes préalablement identifiés, la méthodologie suivie a consisté à identifier les faiblesses de différents modèles exprimés dans le langage déclaratif offert par la programmation par contraintes, puis à proposer des techniques permettant de les combler. Une attention soutenue a été portée aux interactions entre plusieurs composantes essentielles d'une approche en programmation par contraintes : le modèle ; le niveau de filtrage des contraintes ; et les algorithmes de recherche.
Ainsi, nos travaux sur les problèmes d'atelier ont démontré que la plupart des algorithmes de recherche étaient très sensibles au modèle et aux propagateurs choisis. En effet, des variations de performance significatives ont été observées pour des modèles logiquement équivalents en fonction du niveau de filtrage et de l'algorithme de recherche employés. Entre autres, ces expérimentations ont remis en cause l'utilisation systématique de contraintes globales couplées à des propagateurs complexes puisque leur présence a quelquefois dégradé les performances, ou pire, a provoqué un conflit avec l'algorithme de recherche. Par ailleurs, il n'a pas été possible de faire apparaître une unique configuration dominante, mais plutôt un ensemble restreint de très bonnes configurations.
Après un travail préliminaire de modélisation de notre problème de fournées, nos travaux ont surtout porté sur des raisonnements liés à l'optimalité des solutions. En effet, la structure de ce problème entraîne une mauvaise évaluation de la borne inférieure par le solveur lors de la résolution et on peut souvent observer une augmentation brutale de celle-ci lors de la découverte d'une solution. Nous avons montré que l'intégration de règles de filtrage basées sur les coûts dans un algorithme de recherche classique améliorait grandement la résolution du problème de fournées traité dans cette thèse. En effet, une meilleure borne inférieure permet d'éviter l'exploration de larges pans de l'espace de recherche.
Concernant nos contributions sur le solveur choco, nous avons surtout cherché à promouvoir les techniques proposées au cours de cette thèse. Cependant, ce travail a engendré des réflexions et des contributions plus générales sur le langage de modélisation, l'architecture interne du solveur, ou encore sa flexibilité. Nous avons affronté les difficultés issues de la cohabitation au sein du solveur de techniques redondantes, ou même contradictoires. En effet, la simplicité du langage déclaratif de modélisation et le principe de la boîte noire cachent les mécanismes internes et l'implémentation des solveurs. Si ce choix a des avantages certains pour les nouveaux utilisateurs ou lors de la résolution de problème simple, il peut devenir un obstacle à la résolution de problèmes complexes pour lesquels une compréhension approfondie du fonctionnement du solveur peut s'avérer nécessaire. Ainsi, nous avons justifié nos choix d'implémentation et nous proposons plusieurs outils de reformulation et de configuration pour comparer les modèles, niveaux de filtrage et algorithmes de recherche.
Tout au long de cette thèse, nous avons discuté de certaines limitations et perspectives concernant les approches proposées. Nous rappelons maintenant les perspectives les plus pertinentes en les envisageant d'un point de vue global.

10.1 Problèmes d'atelier

Les premières perspectives consistent à améliorer certains aspects de notre approche. Tout d'abord, l'incorporation d'un mécanisme de guidage des heuristiques d'arbitrage basé sur les dernières solutions découvertes semble prometteuse, car elle a déjà été appliquée avec succès par d'autres approches en ordonnancement sous contraintes. Ensuite, une contribution importante pourrait être l'intégration des propriétés de Brucker sur les solutions améliorantes dans un algorithme de recherche générique sans passer par une stratégie de branchement dédiée comme ce fut le cas jusqu'à présent.
D'autre part, une perspective immédiate est l'application de notre approche sur d'autres problèmes d'atelier, par exemple avec un autre critère d'optimalité ou avec la présence de temps d'attente minimaux ou maximaux entre les tâches. Cette généralisation permettrait une analyse encore plus fine du comportement des différents modèles, niveaux de filtrage et algorithmes de recherche. Au vu des résultats précédents, les axes prioritaires devraient porter sur l'influence de la randomisation, l'élaboration d'un critère de choix entre les procédures d'optimisation bottom-up et top-down.
Une perspective plus lointaine est de proposer des contraintes globales dont les propagateurs s'adaptent dynamiquement en fonction de l'état des domaines pendant la résolution. En effet, il est fréquent que certaines ressources jouent un rôle prépondérant dans la réalisation d'un projet. Il est parfois possible de les identifier lors de la modélisation, mais leur rôle apparaît quelquefois au cours de la résolution. Dans ce cas, il serait souhaitable que les contraintes soient capables de changer leurs propagateurs en fonction de l'état des domaines. Cependant, il est très difficile d'automatiser le choix des propagateurs et il faut conserver le déterminisme du solveur qui revêt une importante critique pour la plupart des utilisateurs.

10.2 Problème de fournées

Au vu du succès rencontré par le filtrage basé sur les coûts, il serait intéressant d'évaluer des stratégies de branchement basées sur les coûts qui se sont révélées efficaces pour d'autres problèmes. Ensuite, il est sûrement possible d'apporter quelques améliorations algorithmiques aux règles de filtrage proposées dans cette thèse.
Cependant, la perspective la plus intéressante est la généralisation de la relaxation et des règles de filtrage associées à tous les problèmes de fournées sans contraintes de précédence ni dates de disponibilité dont les homologues à une machine sont résolus en temps polynomial. Les difficultés majeures concerneront la généralisation de la règle de filtrage basé sur le coût du nombre de fournées non vides et l'adaptation de l'algorithme de filtrage du placement des tâches.
Une perspective à long terme concerne le traitement des problèmes de machines à traitement par fournées parallèles.

10.3 Solveur choco

Concernant le solveur `choco`, notre première perspective est d'intégrer les éléments de documentation présents dans cette thèse à la documentation officielle de `choco`. De plus, il faudrait tester intensivement les modules de reformulation et de prétraitement sur des problèmes moins structurés que les problèmes d'atelier.
Un axe majeur du développement concerne les problèmes à machine parallèle avec allocation de ressources. On peut distinguer deux problématiques principales au niveau du solveur que sont l'intégration totale des domaines hypothétiques et la gestion de contraintes globales multi-ressources. Les contraintes transversales définissant l'usage des ressources par les tâches sont encore propagées par une surcouche du solveur et non par le moteur de propagation. Ensuite, la modélisation de l'allocation de ressources aux tâches est aisée et flexible, mais l'implémentation souffre d'un grave défaut lorsque le problème est structuré. En effet, une contrainte globale distincte est définie pour chaque ressource avec des tâches optionnelles ce qui entraîne un filtrage redondant lorsque les ressources sont totalement parallèles, c'est-à-dire elles peuvent réaliser initialement les mêmes tâches. Ce phénomène affecte les contraintes de partage de ressource, mais aussi la contrainte de placement à une dimension.

Annexes

Annexe A

Outils choco : développement et expérimentation

Nous décrivons des outils génériques de développement et d'expérimentation disponibles dans le solveur de contraintes choco. Ces modules permettent de reproduire facilement toutes les expérimentations réalisées au cours de cette thèse.

Sommaire

A.1 Procédure générique de traitement d'une instance 119
A.2 Assistant à la création d'un programme en ligne de commande 121
A.3 Intégration dans une base de données 121

Nous introduisons des outils génériques pour le développement et l'évaluation de modèle(s) choco. Ces outils sont basés uniquement sur des logiciels libres et sont donc livrés avec choco. Ils ont été initialement implémentés pour la compétition de solveurs (*CSP solver competition*) évoquée en section 9.5. Ils ont aussi servi aux évaluations réalisées aux chapitres 7, 8 (les évaluations du chapitre 6 ont été réalisées sur des versions préliminaires de ces outils). L'idée directrice est de proposer un ensemble cohérent de services pour la résolution d'une instance d'un CSP (section A.1), la création d'un programme en ligne de commande pour lancer et configurer la résolution d'une ou plusieurs instances (section A.2), et stocker les résultats obtenus dans plusieurs formats facilitant leur analyse numérique (section A.3).
Nous apportons régulièrement des améliorations à ces outils et espérons voir leur utilisation se généraliser. Un exemple d'utilisation sur un problème de *bin packing* est présenté en annexe B.3.

A.1 Procédure générique de traitement d'une instance

La classe abstraite `AbstractInstanceModel` propose une procédure générique de traitement d'une instance de CSP. La classe `AbstractMinimizeModel` spécialise cette dernière pour les problèmes de minimisation. Un tel objet est réutilisable (on peut résoudre différentes instances de CSP sans le recréer) et déterministe (on peut reproduire une exécution en spécifiant une graine pour le générateur de nombres aléatoires). Nous nous appuierons sur le listing 2 pour détailler ses principales fonctionnalités.

Listing 2 – Méthode principale de la classe `AbstractInstanceModel`

```
1 public final void solveFile(File file) {
2 initialize();
3 try {
4   LOGGER.log(Level.CONFIG, INSTANCE_MSG, file.getName());
5   boolean isLoaded = false;
6   time[0] = System.currentTimeMillis();
7   try {
```

```
8       load(file);
9       isLoaded = true;
10    } catch (UnsupportedConstraintException e) {
11      Arrays.fill(time, 1, time.length, time[0]);
12      logOnError(UNSUPPORTED, e);
13    }
14    if( isLoaded) {
15      LOGGER.config("loading...[OK]");
16      time[1] = System.currentTimeMillis();
17      isFeasible = preprocess();
18      status = postAnalyzePP();
19      initialObjective = objective;
20      logOnPP();
21      time[2] = System.currentTimeMillis();
22      if( applyCP() ) {
23        //try to solve the problem using CP.
24        model = buildModel();
25        logOnModel();
26        time[3] = System.currentTimeMillis();
27        solver = buildSolver();
28        logOnSolver();
29        time[4] = System.currentTimeMillis();
30        //isFeasible is either null or TRUE;
31        if( isFeasible == Boolean.TRUE) solve();
32        else isFeasible = solve();
33        time[5] = System.currentTimeMillis();
34        status = postAnalyzeCP();
35      }else {
36        //preprocess is enough to determine the instance status
37        Arrays.fill(time, 3, time.length, time[2]);
38      }
39      //check the solution, if any
40      if( isFeasible == Boolean.TRUE) {
41        checkSolution();
42        LOGGER.config("checker...[OK]");
43      }
44      //reporting
45      makeReports();
46    }
47
48  } catch (Exception e) {
49    logOnError(ERROR, e);
50  }
51  ChocoLogging.flushLogs();
52  }
```

Après l'initialisation, une instance d'un CSP est généralement chargée (ligne 8) par la lecture d'un fichier donné en paramètre. Cependant, on peut aussi définir un générateur aléatoire d'instance. En cas de succès du chargement, on procède au prétraitement des données (ligne 17) nécessaire à la construction du modèle (tris, calculs intermédiaires, bornes inférieures, bornes supérieures). Le prétraitement peut éventuellement renvoyer une solution dont l'analyse détermine le statut de la résolution. Si nécessaire, on procède alors à la construction du modèle (ligne 24) qui définit les variables et les contraintes puis à celle du solveur (ligne 27) qui définit l'algorithme de recherche, la stratégie de branchement, et les algorithmes de filtrage. On lance ensuite la résolution (ligne 32) dont le résultat est analysé. En présence d'une solution, choco applique un processus de vérification automatique des solutions indépendant des algorithmes de filtrage (ligne 41). Notez qu'il est maintenant possible de vérifier toutes les solutions trouvées par le solveur en activant les assertions Java (option `-ea`). On peut aussi utiliser des outils externes comme le vérificateur de la CSC'2009 ou un fichier de `Properties` contenant les meilleurs résultats connus sur les instances.

Finalement, on rédige différents comptes-rendus sur la résolution (ligne 45). L'utilisateur peut définir ses propres rapports, par exemple un affichage ou une analyse de la solution, en surchargeant cette méthode. La méthode `solveFile(File)` intercepte toutes les exceptions Java pour éviter une interruption brutale de la résolution qui disqualifie des compétitions de solveurs.

La présence d'instructions de la classe `java.util.Logging` témoigne du fait que la classe `AbstractInstanceModel` est intégrée au système de logs de choco (la sortie par défaut est la console). La syntaxe des messages respecte le formalisme de la *CSP solver competition*. À cette occasion, nous avons remanié le système de logs qui est devenu non intrusif ce qui a entraîné une amélioration significative des performances de choco. L'architecture de ce système et la syntaxe des messages (selon la verbosité) sont décrites précisément dans la documentation officielle.

A.2 Assistant à la création d'un programme en ligne de commande

La classe abstraite `AbstractBenchmarkCmd` facilite la création de programmes en ligne de commande pour lancer et configurer la résolution d'une ou plusieurs instances par un objet héritant de `AbstractInstanceModel`. Cette classe est particulièrement utile lorsque les évaluations sont réalisées sur une grille de calcul puisqu'il est alors nécessaire de disposer d'une ligne de commande. Cette classe s'appuie sur la librairie args4j qui propose la lecture des arguments d'une ligne de commandes à partir d'annotations Java.
Notre classe définit quelques options pour la configuration d'un objet `AbstractInstanceModel` pour : (a) gérer les entrées (fichiers d'instance) et les sorties (comptes-rendus), (b) sélectionner un fichier `.properties` contenant la `Configuration` du solveur, (c) assurer le déterminisme de la commande (générateur de nombres aléatoires), et (d) créer une connexion avec la base de données. Si l'argument de l'option obligatoire `-f` est un chemin de fichier, alors on résout cet unique fichier d'instance. Si la valeur est un chemin vers un répertoire, alors on explore récursivement ce répertoire en traitant certains fichiers. Un fichier est traité s'il correspond à un des motifs (*pattern*) définis dans un nombre quelconque d'arguments supplémentaires de la commande. Les jokers (*wildcards*) sont autorisés dans les motifs.
Un écran d'aide de la commande est généré automatiquement à partir de ces annotations :

```
the available options are:
Example of command with required arguments:
java [-jar myCmd.jar| -cp myCmd.jar MyCmd.class]  -f (--file, -file) FILE
Example of command with all arguments:
java [-jar myCmd.jar| -cp myCmd.jar MyCmd.class]  -e (--export) FILE -f (--file, -file) FILE
  -o (--output) FILE -p (--properties) FILE -s (--seed, -seed) N -tl (--timeLimit, -time) N
  -u (--url) VAL -v (--verbosity) [OFF|SILENT|QUIET|DEFAULT|VERBOSE|SOLUTION|SEARCH|FINEST]
Options:
 -e (--export) FILE                    : activate embedded database and export it to odb file
 -f (--file, -file) FILE               : Instance File or directory with option
                                         al wildcard pattern arguments.
 -o (--output) FILE                    : specify output directory (logs, solutions, ...)
 -p (--properties) FILE                : user properties file
 -s (--seed, -seed) N                  : global seed
 -tl (--timeLimit, -time) N            : time limit in seconds
 -u (--url) VAL                        : connect to remote database at URL.
 -v (--verbosity) [OFF | SILENT | QUIET : set the verbosity level
  | DEFAULT | VERBOSE | SOLUTION | SEAR :
 CH | FINEST]                          :
```

A.3 Intégration dans une base de données

Pour finir, nous avons défini une base de données relationnelle en Java basée sur hsqldb qui permet de stocker les résultats de la résolution d'un objet `Solver` ou `AbstractInstanceModel`. Le solveur communique avec la base de données grâce à l'interface Java Database Connectivity (JDBC). Le format de fichier du logiciel oobase de la suite Open Office est une archive contenant une base de données hsqldb

et divers autres fichiers propres oobase. Nous avons développé un modèle oobase contenant une base de données vide et plusieurs rapports et formulaires prédéfinis. Une commande (section A.2) peut générer un fichier oobase basé sur ce modèle contenant les résultats de son exécution. On peut préférer communiquer les résultats à un serveur hsqldb pour agréger les résultats de plusieurs commandes, par exemple obtenus dans une grille de calcul. Il incombe alors à l'utilisateur de lancer le serveur (droit administrateur) et y extraire la base de données du modèle oobase.

Annexe B

Tutoriel choco : ordonnancement et placement

Ce tutoriel introduit les bases pour l'utilisation des modules de choco décrits au chapitre 9. Les deux premiers cas d'utilisation couvrent un large spectre du module d'ordonnancement, et le dernier se concentre sur le module de développement et d'expérimentation à travers la résolution d'un problème de placement en une dimension. La difficulté des cas d'utilisation suit une progression croissante.

Sommaire

B.1 Gestion de projet : construction d'une maison . . . 123
B.1.1 Approche déterministe . . . 124
B.1.2 Approche probabiliste . . . 126
B.2 Ordonnancement cumulatif . . . 129
B.3 Placement à une dimension . . . 132
B.3.1 Traitement d'une instance . . . 132
B.3.2 Création d'une commande . . . 134
B.3.3 Cas d'utilisation . . . 135

Ce chapitre présente trois tutoriels sur les fonctionnalités de choco présentées en chapitre 9. Les sections B.1 et B.2 décrivent l'utilisation du module d'ordonnancement sous contraintes. Ces deux tutoriels sont basés sur l'interface `samples.tutorials.Example` qui définit un schéma de traitement simple des exemples. La section B.3 décrit l'utilisation des outils de développement et d'expérimentation pour la résolution d'un problème de placement. Ce dernier tutoriel est plus avancé puisqu'il aborde d'autres sujets que la modélisation.

B.1 Gestion de projet : construction d'une maison

Ce tutoriel décrit la gestion d'un projet [181] de construction d'une maison avec choco. Un projet est défini [ISO10006, 1997] comme :

> « un processus unique, qui consiste en un ensemble d'activités coordonnées et maîtrisées comportant des dates de début et de fin, entreprises dans le but d'atteindre un objectif conforme à des exigences spécifiques telles que des contraintes de délais, de coûts et de ressources. »

Les grandes phases d'un projet sont :

Avant-projet passer de l'idée initiale au projet formalisé elle aboutit à la décision du maître d'ouvrage de démarrer ou non le projet.

Définition mise en place du projet, de l'organisation du projet et des outils de gestion associés.

Réalisation superviser l'exécution des différentes tâches nécessaires à la réalisation du projet et gérer les modifications qui apparaissent au fur et à mesure de la réalisation.

Négliger la définition d'un projet est une erreur car sa réalisation va forcément prendre du retard s'il n'est pas correctement défini au départ. En effet, La caractéristique d'irréversibilité forte d'un projet signifie que l'on dispose d'une capacité d'action très forte au début, mais au fur et à mesure de l'avancement du projet et des prises de décision, la capacité d'action diminue, car les choix passés limitent les possibilités d'action en fin de projet.
La gestion d'un projet est souvent évaluée en fonction du dépassement de coûts et de délais ainsi que sur la qualité du produit fini. On considère souvent une part d'incertitude sur la réalisation des tâches ou due à l'environnement extérieur.
Les statistiques réunies par le *Standish Group* concernant les États-Unis en 1998 révèlent que : (a) seulement 44% des projets sont achevés dans les temps, (b) les projets dépassent en moyenne leur budget de 189%, (c) 70% des projets ne se réalisent pas comme prévu, (d) les projets sont en moyenne 222% plus long que prévu. Nous nous intéressons à deux méthodes de gestion de projet très répandues qui améliorent ces performances :

Programme Evaluation Review Technique (PERT) introduit par l'armée américaine (navy) en 1958 pour contrôler les coûts et l'ordonnancement de la construction des sous-marins Polaris.

Critical Path Method (CPM) introduit par l'industrie américaine en in 1958 (DuPont Corporation and Remington-Rand) pour contrôler les coûts et l'ordonnancement de production.

Les méthodes PERT/CPM ignorent la plupart des dépendances, c'est-à-dire qu'elles ne considèrent que les contraintes de précédences. On ignore donc les contraintes de coûts, partage de ressources ou d'éventuelles préférences. Par contre, elles permettent de répondre aux questions suivantes.

- Quelle sera le délai total du projet ? Quels sont les risques ?
- Quelles sont les activités critiques qui perturberont l'ensemble du projet si elles ne sont pas achevées dans les délais ?
- Quel est le statut du projet ? En avance ? Dans les temps ? En retard ?
- Si la fin du projet doit être avancée, quelle est la meilleure manière de procéder ?

Les méthodes PERT/CPM sont disponibles dans de nombreux logiciels de gestion de projet généralement par le biais d'algorithmes dédiés. Nous montrons d'abord qu'il est plus facile d'appliquer ces méthodes grâce à un solveur de contraintes. Un autre intérêt de cette approche est la prise en compte de contraintes additionnelles pendant la propagation. On bénéficie aussi de tous les services offerts par le solveur pour le diagnostic, la simulation, la détection des inconsistances, l'explication des conflits, la modélisation d'une partie des coûts et bien sûr une recherche complète ou incomplète pour la satisfaction ou l'optimisation.
Ce tutoriel vous aidera à :

- créer un modèle d'ordonnancement de projet contenant des tâches et des contraintes temporelles (`precedence` et `precedenceDisjoint`),
- construire et visualiser le graphe disjonctif,
- appliquer la méthode PERT/CPM (par propagation) pour la gestion d'un projet.
- donner une intuition de la gestion de contraintes additionnelles en considérant deux gammes opératoires différentes pour la réalisation du projet.

Le code est disponible dans la classe `samples.tutorials.scheduling.PertCPM` qui implémente l'interface `samples.tutorials.Example`.

B.1.1 Approche déterministe

L'approche déterministe ne considère aucune incertitude sur la réalisation des activités. Nous décrivons les étapes classiques de la phase de définition d'un projet et d'un modèle en ordonnancement sous contraintes. L'étape 1 décrit certains champs de la classe. Les étapes 2, 3 et 4 sont réalisées dans la méthode `buildModel()`. L'étape 5 est réalisée dans la méthode `buildSolver()`. Les étapes 6 et 7 sont réalisées dans la méthode `solve()`.
Étape 1 : spécifier l'ensemble des activités.

```
1  private TaskVariable masonry, carpentry, plumbing, ceiling,
    roofing, painting, windows,facade, garden, moving;
```

Étape 2 : spécifier les caractéristiques des activités.
Nous ne considérons aucune incertitude sur la durée des activités. L'horizon de planification est la date

d'achèvement du projet en cas d'exécution séquentielle des tâches (la somme de leurs durées).

```
masonry=makeTaskVar("masonry",horizon, 7);
carpentry=makeTaskVar("carpentry",horizon, 3);
plumbing=makeTaskVar("plumbing",horizon, 8);
ceiling=makeTaskVar("ceiling",horizon, 3);
roofing=makeTaskVar("roofing",horizon, 1);
painting=makeTaskVar("painting",horizon, 2);
windows=makeTaskVar("windows",horizon, 1);
facade=makeTaskVar("facade",horizon, 2);
garden=makeTaskVar("garden",horizon, 1);
moving=makeTaskVar("moving",horizon, 1);
```

Étape 3 : déterminer les contraintes de précédences.

```
model.addConstraints(
  startsAfterEnd(carpentry,masonry),
  startsAfterEnd(plumbing,masonry),
  startsAfterEnd(ceiling,masonry),
  startsAfterEnd(roofing,carpentry),
  startsAfterEnd(roofing,ceiling),
  startsAfterEnd(windows,roofing),
  startsAfterEnd(painting,windows),
  startsAfterEnd(facade,roofing),
  startsAfterEnd(garden,roofing),
  startsAfterEnd(garden,plumbing),
  startsAfterEnd(moving,facade),
  startsAfterEnd(moving,garden),
  startsAfterEnd(moving,painting)
);
```

Étape 4 : déterminer les contraintes additionnelles.
Nous considérons une unique contrainte additionnelle qui définit deux gammes opératoires pour la réalisation des tâches `facade` et `plumbing`. Ces gammes sont définies par une contrainte de disjonction avec temps d'attente `facade` $\sim$ `plumbing`. L'objectif est de déterminer le meilleur arbitrage, ce qui est équivalent, dans ce cas simple, à un arbitrage complet du graphe disjonctif généralisé. Notre approche reste valable lorsqu'il faut arbitrer un petit nombre de disjonctions pour obtenir un arbitrage complet.

```
direction = makeBooleanVar(StringUtils.dirName(facade.getName(), plumbing.getName()));
model.addConstraint(precedenceDisjoint(facade, plumbing, direction, 1, 3));
```

Étape 5 : construire et dessiner le graphe disjonctif généralisé.
Le prétraitement du modèle décrit en section 9.4 est optionnel. Cependant, il permet d'accéder à une interface de visualisation du graphe disjonctif dont les résultats sont donnés en figure B.1.

```
PreProcessCPSolver solver = new PreProcessCPSolver();
PreProcessConfiguration.keepSchedulingPreProcess(solver);
solver.createMakespan();
solver.read(model);

createAndShowGUI(s.getDisjMod());
//DisjunctiveSModel disjSMod = new DisjunctiveSModel(solver);
//createAndShowGUI(disjSMod);
```

Étape 6 : estimation initiale des fenêtres de temps des activités.
Cette première estimation présentée en figure B.1(b) est simplement obtenue par la propagation initiale sans prendre de décision d'arbitrage.

```
solver.propagate();
```

Étape 7 : identifier le ou les chemins critiques.
On propage la date de fin au plus tôt de chaque arbitrage. Il existe au moins un ordonnancement au plus tôt réalisable puisque qu'il n'y a ni contraintes de coût, ni contraintes de partage de ressource. On fixe au plus tôt la date d'achèvement du projet (`makespan`), puis on propage cette décision. Remarquez que l'on gère les structures backtrackables de choco avec des appels à `worldPush()` et `worldPop()`.
Un chemin critique est un plus long chemin dans le graphe de précédence. Nous ne détaillons pas le calcul des chemins critiques, par exemple grâce à l'algorithme dynamique de Bellman [37]. Le résultat de cette étape est identique à celui de la méthode PERT/CPM. Les figures B.1(d) et B.1(c) illustrent les résultats obtenus pour les deux arbitrages possibles. On constate que le meilleur arbitrage est `plumbing` $\prec$ `facade`, car le projet finit plus tôt (à la date 21 au lieu de 24) et il y a un unique chemin critique (au lieu de deux).

```
final IntDomainVar dir = solver.getVar(direction);
final IntDomainVar makespan = solver.getMakespan();

solver.worldPush();

dir.instantiate(1, null, true); //set forward
//dir.instantiate(0, null, true); //set backward
solver.propagate();
makespan.instantiate(makespan.getInf(), null, true); //instantiate makespan
solver.propagate(); //compute slack times
createAndShowGUI(disjSMod);

solver.worldPop();
```

Les activités appartenant à un chemin critique, dites *activités critiques*, déterminent à elles seules la date d'achèvement du projet. Ces activités critiques ne peuvent pas être retardées sans retarder l'ensemble du projet. Réciproquement, il est nécessaire de réduire la longueur des chemins critiques pour avancer la fin du projet. Ensuite, l'affectation des personnes et ressources peut être améliorée en se concentrant sur ces activités critiques. Les activités non critiques n'influencent pas la date d'achèvement du projet. On peut alors les réordonnancer et libérer les ressources qui leur sont allouées sans perturber le reste du projet (lissage des ressources).
L'analyse des chemins critiques est donc un aspect important de la gestion de projet. Par exemple, on peut identifier certaines tâches fondamentales (*milestone*) qui sont cruciales pour l'accomplissement du projet. Dans notre cas, la tâche `masonry` est fondamentale puisqu'elle appartient nécessairement à tous les chemins critiques. On doit la réaliser avec une attention approfondie et un contrôle rigoureux.

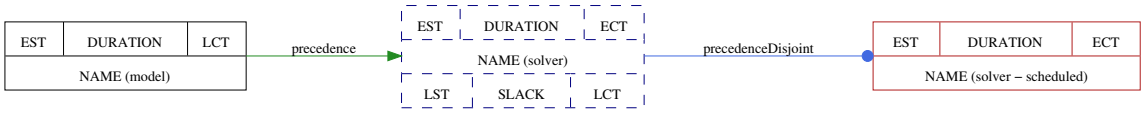
Une solution classique pour avancer un projet est de procéder à du *crashing* qui consiste à réquisitionner des ressources ou utiliser un prestataire pour réaliser plus rapidement une tâche. En général, le *crashing* a un coût et il faut l'utiliser avec précaution.

B.1.2 Approche probabiliste

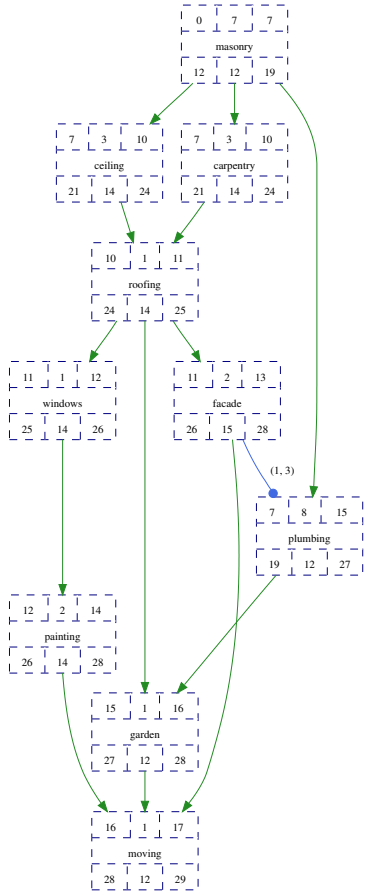
L'approche probabiliste permet de modéliser les incertitudes sur les durées d'exécution des activités. On considère que la durée d'une activité est une variable aléatoire, ce qui signifie qu'elle peut prendre plusieurs valeurs, selon une distribution de probabilité. À partir de ces valeurs, on pourra calculer la durée moyenne, la variance et l'écart type des tâches et des chemins critiques. Si la durée d'une tâche se trouve accrue, la durée du projet peut changer et le chemin critique se déplacer. En gestion de projet, une estimation est généralement (dans les livres et les logiciels) proposée par défaut, elle considère trois durées pour chaque tâche : la durée optimiste p^O, la durée probable (*likely*) p^L et la durée pessimiste p^P. On peut alors calculer l'espérance de la durée p^E, son écart-type σ et sa variance σ^2 grâce aux formules suivantes :

$$p^E = \frac{p^P + 4 \times p^L + p^O}{6} \qquad \sigma = \frac{p^O - p^P}{6}$$

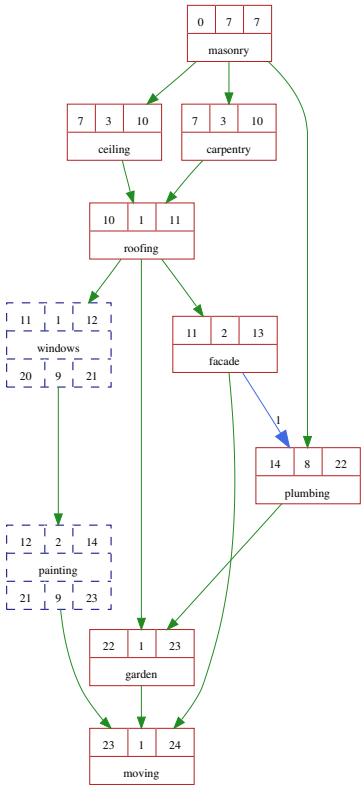
On considère donc que la durée probable se réalise quatre fois plus souvent que les durées optimiste ou pessimiste. Ce ne sera généralement pas le cas, mais ces estimateurs sont sans biais et convergents,



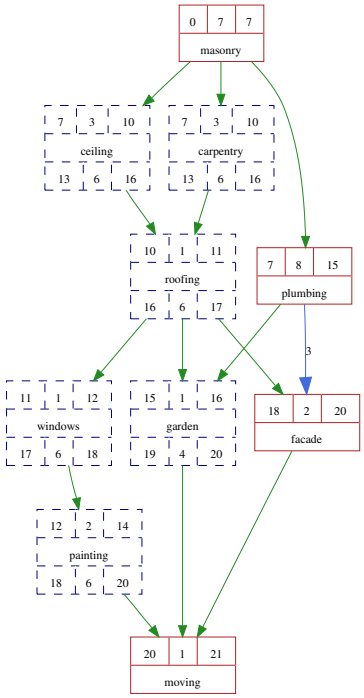
(a) Légende du graphe disjonctif généralisé (modèle et solveur).



(b) Étape 6



(c) Ét. 7 : `facade` ≺ `plumbing`



(d) Ét. 7 : `plumbing` ≺ `facade`

FIGURE B.1 – Visualisation des graphes disjonctifs (activée à l'étape 5) pendant les étapes 6 et 7.

c'est-à-dire que les écarts avec la réalité seront tantôt minorés, tantôt majorés, mais sur un grand nombre de tâches, l'écart sera faible. Le tableau B.1 donne les estimations des durées et variances des tâches. Par un heureux hasard, les durées espérées sont égales aux durées du cas déterministe et la figure B.1 illustre les résultats de la méthode PERT/CPM. On peut calculer directement les durées espérées dans le modèle en substituant des variables entières définies à l'aide de la formule précédente aux constantes dans l'étape 2. La longueur d'un chemin S est égale à la somme des durées des tâches du chemin. Dans

Tâche	p^O	p^L	p^P	p^E	σ^2
masonry	2	6	16	7	5.44
carpentry	1	2	9	3	1.77
plumbing	4	8	12	8	1.77
ceiling	1	3	5	3	0.44
roofing	0	1	2	1	0.11
painting	1	1	7	2	1
windows	0	1	2	1	0.11
facade	1	1	7	2	1
garden	0	1	2	1	0.11
moving	1	1	1	1	0

TABLE B.1 – Estimation probabiliste des durées des tâches.

le cas probabiliste, la somme des durées S est une variable aléatoire qui peut prendre plusieurs valeurs selon une distribution de probabilité. Si le nombre de tâches du chemin est grand ($n > 30$), la loi suivie par la somme tend vers une loi normale de moyenne S et de variance V où V est la somme des variances sur le chemin critique (théorème central limite).

Arbitrage	Chemin critique	S	V
`facade` $\prec$ `plumbing`	`masonry, facade, plumbing, moving`	21	8.21
`plumbing` $\prec$ `facade`	`masonry, ceiling, roofing, facade, plumbing, garden, moving`	24	8.87
`plumbing` $\prec$ `facade`	`masonry, carpentry, roofing, facade, plumbing, garden, moving`	24	10.2

TABLE B.2 – Longueurs et variances des chemins critiques.

Le tableau B.2 récapitule les longueurs et variances des chemins critiques pour les deux arbitrages possibles. Une erreur est introduite dans le calcul de la durée moyenne et de la variance du projet entier, car nous ne considérons qu'un seul chemin critique sans tenir compte d'aucun autre chemin. Ces erreurs varient en fonction de la configuration du réseau. La date d'achèvement du projet est influencée de manière optimiste, mais l'écart-type peut être influencé dans les deux sens. Si un graphe possède un unique chemin critique dont la longueur est nettement supérieure à celles des autres chemins, l'erreur sera acceptable. Au contraire, si un graphe possède de nombreux chemins critiques ou quasi critiques, l'erreur sera plus importante. Cependant, si les chemins en question possèdent un grand nombre d'activités communes, l'erreur introduite sera amoindrie.
La probabilité que le projet finisse au plus tard à la date D est calculée en se référant à la probabilité correspondant à Z dans une table standard de la distribution normale :

$$Z = \frac{D - S}{\sqrt{V}}$$

Les deux courbes de la figure B.2 représentent ces probabilités en considérant seulement un chemin critique dont la variance est maximale pour chaque arbitrage (lignes 1 et 3 du tableau B.2). On constate sans surprise que l'arbitrage `facade` $\prec$ `plumbing` reste le meilleur.

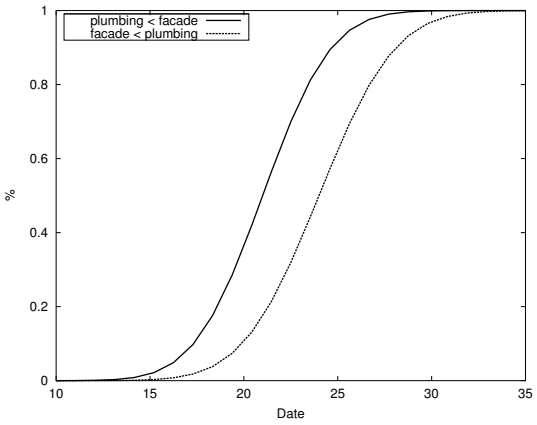


FIGURE B.2 – Probabilité que le projet finisse au plus tard à une date D.

B.2 Ordonnancement cumulatif

Nous présentons ici un problème d'ordonnancement cumulatif simple résolu à l'aide de la contrainte `cumulative`. Ce problème consiste à ordonnancer un nombre maximal de tâches sur une ressource cumulative alternative avec un horizon fixé. Une tâche optionnelle qui a été éliminée de la ressource n'appartient pas à l'ordonnancement final.
La figure B.3 décrit l'instance utilisée à titre d'exemple. À gauche, nous donnons le profil suivi par la capacité de la ressource sur notre horizon de planification. À droite, les tâches sont représentées par des rectangles dont la largeur est la durée d'exécution de la tâche, et la longueur est la consommation instantanée (hauteur) de la tâche sur la ressource. Ce tutoriel vous aidera à :

- poser deux modèles logiquement équivalents basés sur une contrainte globale `cumulative` régulière ou alternative,
- modéliser une capacité variable dans le temps alors que les constructeurs de la contrainte cumulative acceptent uniquement une capacité fixe,
- utiliser une fonction objectif dépendant de l'exécution ou non des tâches optionnelles,
- configurer une stratégie de recherche qui alloue les tâches à la ressource, puis les ordonnance.

Le code est disponible dans la classe `samples.tutorials.scheduling.CumulativeScheduling` qui implémente l'interface `samples.tutorials.Example`. L'étape 1 décrit certains champs de la classe. Les étapes 2, 3, 4 et 5 sont réalisées dans la méthode `buildModel()`. Les étapes 6 et 7 sont respectivement réalisées dans les méthodes `buildSolver()` et `solve()`. L'étape 8 est réalisée dans la méthode

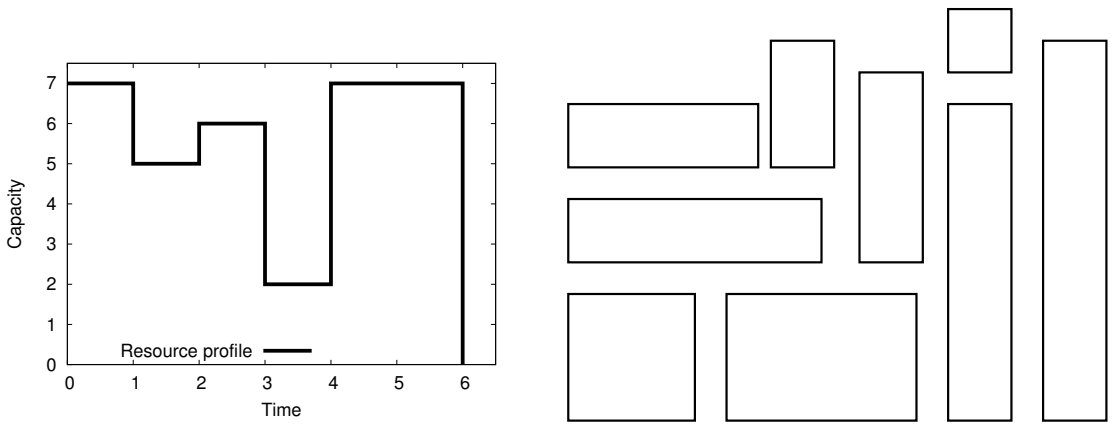


FIGURE B.3 – Une instance de notre problème d'ordonnancement cumulatif.

prettyOut().
Étape 1 : modéliser le profil de la ressource.
Il suffit de fixer la capacité de la ressource à sa valeur maximale sur l'horizon et d'ordonnancer des tâches fictives simulant la diminution temporaire de la capacité. Dans notre cas, nous avons besoin de trois (NF) tâches fictives de durées unitaires et de hauteurs égales à 2, 1 et 4. On considère N tâches dont NF tâches fictives et NT tâches réelles. Les durées et hauteurs des tâches sont contenues dans des objets int[] où les tâches fictives correspondent aux NF premiers éléments.

```
protected final static int NF = 3, NT = 11, N = NF + NT;

private final static int[] DURATIONS = new int[]{1, 1, 1, 2, 1, 3, 1, 1, 3, 4, 2, 3, 1, 1};
private final static int HORIZON = 6;

private final static int[] HEIGHTS = new int[]{2, 1, 4, 2, 3, 1, 5, 6, 2, 1, 3, 1, 1, 2};
private final static int CAPACITY = 7;
```

Étape 2 : définir les variables du modèle.
On définit N tâches avec des fenêtres de temps initiales identiques $[0, \texttt{HORIZON}]$. On impose que les domaines des variables entières représentant la tâche soient bornés (V_BOUND) car la gestion d'un domaine énumérée est plus coûteuse et de surcroît inutile puisque la contrainte cumulative ne raisonne que sur les bornes. On définit ensuite NT variables booléennes qui indiquent si les tâches sont *effectivement* exécutées sur la ressource, c'est-à-dire appartiennent à l'ordonnancement final. La variable usages$[i]$ est égale à 1 si la tâche tasks$[i + \texttt{NF}]$ est effective. Finalement, on définit la variable objectif (V_BOUND) avec un domaine borné. La définition des hauteurs n'apparaît pas encore, car elle varie en fonction de l'utilisation d'une ressource régulière ou alternative.

```
//the fake tasks to establish the profile capacity of the ressource are the NF firsts.
TaskVariable[] tasks = makeTaskVarArray("T", 0, HORIZON, DURATIONS, V_BOUND);
IntegerVariable[] usages = makeBooleanVarArray("U", NT);
IntegerVariable objective = makeIntVar("obj", 0, NT, V_BOUND, V_OBJECTIVE);
```

Étape 3 : établir le profil de la ressource.
Il suffit d'ordonnancer les tâches fictives pour simuler le profil de la ressource.

```
model.addConstraints(
startsAt(tasks[0], 1),
startsAt(tasks[1], 2),
startsAt(tasks[2], 3)
);
```

Étape 4 : modéliser la fonction objectif.
La valeur de l'objectif est simplement la somme des variables booléennes usages.

```
model.addConstraint(eq( sum(usages), objective));
```

Étape 5 : poser la contrainte de partage de ressource cumulative.
On distingue deux cas en fonction de la valeur du paramètre booléen useAlternativeResource.

vrai on pose une contrainte cumulative alternative en définissant simplement des hauteurs et une capacité constantes. La ressource alternative offre plusieurs avantages : faciliter la modélisation, proposer un filtrage dédié aux tâches optionelles, la gestion d'allocation multi-ressources (useResources).

faux on profite du fait que la contrainte cumulative accepte des hauteurs variables pour simuler l'exécution effective ou l'élimination des tâches. Les tâches fictives gardent une hauteur constante mais les autres variables de hauteur ont un domaine énuméré contenant deux valeurs : zéro et la hauteur. L'astuce consiste à poser les contraintes de liaison suivantes : usages$[i] = 1 \Leftrightarrow$ heights$[i - \texttt{NF}] =$ HEIGHTS$[i]$.

```
Constraint cumulative;
if(useAlternativeResource) {
```

```
heights = constantArray(HEIGHTS);
cumulative = cumulativeMax("alt-cumulative", tasks, heights, usages, constant(CAPACITY),
    NO_OPTION);
}else {
heights = new IntegerVariable[N];
//post the channeling to know if the task uses the resource or not.
for (int i = 0; i < NF; i++) {
  heights[i] = constant(HEIGHTS[i]);
}
for (int i = NF; i < N; i++) {
  heights[i] = makeIntVar("H_" + i, new int[]{0, HEIGHTS[i]});
  model.addConstraint(boolChanneling(usages[i- NF], heights[i], HEIGHTS[i]));
}
cumulative =cumulativeMax("cumulative", tasks, heights, constant(CAPACITY), NO_OPTION);
}
model.addConstraint(cumulative);
```

Étape 6 : configurer le solveur et la stratégie de branchement.
On lit le modèle puis configure le solveur pour une maximisation avant de définir la stratégie de branchement. Par sécurité, on commence par supprimer tout résidu de stratégie. On définit ensuite un premier objet de branchement binaire qui alloue une tâche (branche gauche) puis l'élimine (branche droite) en suivant l'ordre lexicographique des tâches (les indices). À la fin de ce branchement, une allocation des tâches sur la ressource est fixée. Le second branchement (n-aire) ordonnance les tâches en instanciant leurs dates de début avec les heuristiques de sélection dom-minVal. Cette stratégie de branchement simple devrait certainement être adaptée et améliorée pour résoudre un problème réel. Par exemple, on pourrait d'abord allouer les tâches pour lesquelles l'occupation de la ressource est minimale ou éliminer dynamiquement des ordonnancements dominés.

```
solver = new CPSolver();
solver.read(model);

StrategyFactory.setNoStopAtFirstSolution(solver);
StrategyFactory.setDoOptimize(solver, true); //maximize

solver.clearGoals();
solver.addGoal(BranchingFactory.lexicographic(solver, solver.getVar(usages), new MaxVal()));
IntDomainVar[] starts = VariableUtils.getStartVars(solver.getVar(tasks));
solver.addGoal(BranchingFactory.minDomMinVal(solver, starts));
solver.generateSearchStrategy();
```

Étape 7 : lancer la résolution.
La variable objectif a été indiquée par le biais d'une option du modèle et la configuration du solveur réalisée à l'étape précédente. Les deux modèles obtiennent les mêmes résultats, car l'algorithme de filtrage par défaut raisonne aussi bien sur les hauteurs que les usages. Ceci n'est généralement pas le cas pour les raisonnements énergétiques. La découverte d'une première solution avec neuf tâches effectives, puis la preuve de son optimalité sont réalisées en moins d'une seconde en explorant 458 nœuds et en effectuant 998 backtracks. Notre approche est meilleure que la stratégie par défaut qui découvre 10 solutions en explorant 627 nœuds et effectuant 2114 backtracks. Une cause probable de cette amélioration est que la stratégie par défaut de `choco` emploie l'heuristique minVal. Par conséquent, elle élimine d'abord la tâche de la ressource avant d'essayer de l'allouer ce qui semble contradictoire avec notre objectif.

```
solver.launch();
```

Étape 8 : visualiser la solution.
On peut visualiser les contraintes de partage de ressource grâce à la librairie `jfreechart`. On propose un mode de visualisation interactive ou d'export dans différents formats de fichier (`.png`, `.pdf`). La figure B.4 montre la solution optimale de notre instance avec une palette de couleurs monochrome. Les tâches T_{10} et T_{13} n'appartiennent pas à l'ordonnancement final.

```
1  createAndShowGUI(title, createCumulativeChart(title, (CPSolver) solver, cumulative, true));
```

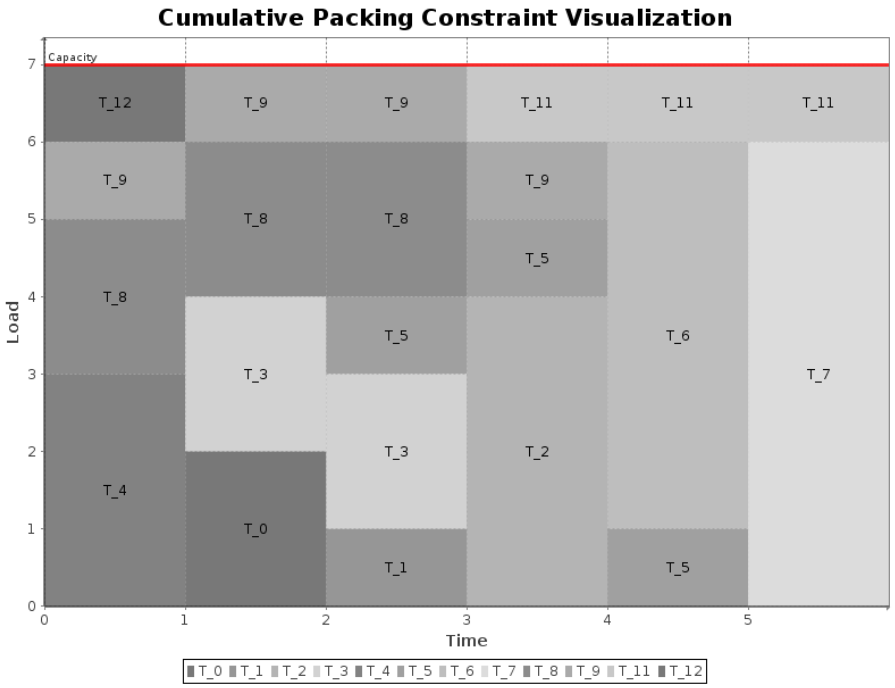


FIGURE B.4 – Une solution optimale de l'instance décrite en figure B.3.

B.3 Placement à une dimension

Ce tutoriel décrit la résolution d'un problème de placement à une dimension (*bin packing*) grâce aux outils de développement et d'expérimentation présentés en chapitre A. On cherche un rangement d'un ensemble d'articles I caractérisés par leur longueur entière positive s_i dans un nombre minimal de conteneurs caractérisés par leur capacité C ($s_i \leq C$). Ce tutoriel vous aidera à :

- créer un modèle pour le placement basé sur la contrainte `pack`,
- appliquer différentes stratégies de recherche,
- étendre la classe `AbstractInstanceModel` pour répondre à vos besoins,
- créer une ligne de commande pour lancer la résolution d'une ou plusieurs instances,
- se connecter à une base de données par le biais de la ligne de commande.

Le code est disponible dans le package `samples.tutorials.packing.parser`.

B.3.1 Traitement d'une instance

Dans cette section, nous décrivons la classe `BinPackingModel` qui hérite de la classe abstraite `AbstractMinimizeModel`. Cette dernière spécialise la classe abstraite `AbstractInstanceModel` (voir section A.1) pour les problèmes de minimisation. La valeur de la propriété booléenne `LIGHT_MODEL` de la classe `Configuration` déterminera la construction d'un modèle *Light* ou *Heavy*.

Étape 1 : définir un parseur pour les fichiers d'instances.

Cet objet implémenté l'interface `InstanceFileParser` qui définit les services requis par `AbstractInstanceModel`. Dans notre cas, une instance est encodée dans un fichier texte dont les première et seconde lignes contiennent respectivement le nombre d'articles et la capacité des conteneurs alors que les suivantes contiennent les longueurs des articles.

Étape 2 : définir le prétraitement de l'instance.

On récupère le parseur pour construire un objet appliquant les heuristiques *complete decreasing best fit*

et *complete decreasing first fit*. Lorsque l'heuristique renvoie une solution, on calcule une borne inférieure (utilisée dynamiquement par pack) afin de tester l'optimalité de la solution heuristique après le prétraitement. Les heuristiques et bornes inférieures utilisées sont disponibles dans choco.

```
@Override
public Boolean preprocess() {
final BinPackingFileParser pr = (BinPackingFileParser) parser;
heuristics = new CompositeHeuristics1BP(pr);
Boolean b = super.preprocess();
if(b != null && Boolean.valueOf(b)) {
  computedLowerBound = LowerBoundFactory.computeL_DFF_1BP(pr.sizes, pr.capacity, heuristics.
      getObjectiveValue().intValue());
  nbBins = heuristics.getObjectiveValue().intValue() - 1;
}else {
  computedLowerBound = 0;
  nbBins = pr.sizes.length;
}
return b;
}
```

Étape 3 : construire le modèle.
Le modèle repose entièrement sur la contrainte pack et les services de l'objet `PackModel`. Le modèle *Heavy* pose des contraintes supplémentaires d'élimination des symétries par rapport au modèle *Light*. Finalement, nous précisons la variable objectif, c'est-à-dire le nombre de conteneurs non vide, définie automatiquement par `PackModel`.

```
@Override
public Model buildModel() {
CPModel m = new CPModel();
final BinPackingFileParser pr = (BinPackingFileParser) parser;
modeler = new PackModel("", pr.sizes, nbBins, pr.capacity);
pack = Choco.pack(modeler, C_PACK_AR, C_PACK_DLB, C_PACK_LBE);
m.addConstraint(pack);
if( ! defaultConf.readBoolean(LIGHT_MODEL) ) {
  pack.addOption(C_PACK_FB);
  m.addConstraints(modeler.packLargeItems());
}
modeler.nbNonEmpty.addOption(V_OBJECTIVE);
return m;
}
```

Étape 4 : construire et configurer le solveur.
Les variables de décision sont les variables d'affectation d'un article dans un conteneur. Le modèle *Heavy* utilise une stratégie de branchement *complete decreasing best fit* avec élimination dynamique de symétries lors d'un retour arrière (voir section 8.3). Le modèle *Light* utilise une stratégie de branchement lexicographique équivalente à *complete decreasing first fit* sans élimination des symétries.

```
@Override
public Solver buildSolver() {
Solver s = super.buildSolver(); // create the solver
s.read(model); //read the model
s.clearGoals();
if(defaultConf.readBoolean(BasicSettings.LIGHT_MODEL) ) {
  s.addGoal(BranchingFactory.lexicographic(s, s.getVar(modeler.getBins())));
}else {
  final PackSConstraint ct = (PackSConstraint) s.getCstr(pack);
  s.addGoal(BranchingFactory.completeDecreasingBestFit(s, ct));
}
s.generateSearchStrategy();
```

```
return s;
}
```

Étape 5 : personnaliser les logs et la visualisation.
On utilise la contrainte `pack` pour afficher textuellement la solution dans les logs. De même, on crée un objet `JFreeChart` pour visualiser le rangement (interactif ou export). L'affichage et la visualisation sont désactivés si la solution optimale est obtenue de manière heuristique. Nous verrons comment régler les logs et la visualisation en section B.3.3.

```
@Override
public String getValuesMessage() {
if(solver != null && solver.existsSolution()) {
  return ( (PackSConstraint) solver.getCstr(pack) ).getSolutionMsg();
} else return "";
}

@Override
public JFreeChart makeSolutionChart() {
return solver != null && solver.existsSolution() ?
    ChocoChartFactory.createPackChart(getInstanceName()+" : "+getStatus(), (PackSConstraint)
        solver.getCstr(pack)) : null;
}
```

B.3.2 Création d'une commande

Dans cette section, nous décrivons la classe `BinPackingCmd` qui hérite de la classe abstraite `AbstractBenchmarkCmd` (voir section A.2).
Étape 6 : définir une nouvelle option pour la ligne de commande.
On associe une annotation Java à un champ de la classe.

```
@Option(name="-l",aliases={"--light"},usage="set the light model")
 protected boolean lightModel;
```

Étape 7 : vérifier les arguments de la ligne de commande.
On charge aussi un fichier `.properties` contenant tous les optimums de notre jeu d'instances.

```
 @Override
protected void checkData() throws CmdLineException {
super.checkData();
seeder = new Random(seed);
//check for Boolean, if null then keep default setting (property file)
if(lightModel) settings.putTrue(BasicSettings.LIGHT_MODEL);
//load status checkers
SCheckFactory.load("/bin-packing-tut/bin-packing-tut.properties");
  }
```

Étape 8 : spécifier comment construire une instance réutilisable.

```
@Override
protected AbstractInstanceModel createInstance() {
     return new BinPackingModel(settings);
}
```

Étape 9 : définir le traitement d'une instance.

```
 @Override
 public boolean execute(File file) {
      instance.solveFile(file);
      return instance.getStatus().isValidWithOptimize();
```

```
5   }
```

Étape 10 : définir la méthode `main(String[])`.
La méthode a toujours la même structure, mais on doit adapter la classe de `cmd`.

```
1   public static void main(String[] args) {
    final BinPackingCmd cmd = new BinPackingCmd();
    if (args.length == 0) {
          cmd.help();
5    } else {
          cmd.doMain(args);
    }
    }
```

B.3.3 Cas d'utilisation

Nous avons sélectionné un jeu de 36 instances avec 50 ou 100 articles issues de http://www.wiwi.uni-jena.de/Entscheidung/binpp/index.htm.
Étape 11 : résoudre une instance et visualiser la solution.

```
java -ea -cp $CHOCO_JAR $CMD_CLASS -v VERBOSE --properties bp-chart.properties
     -f instances/N1C3W1_A.BPP;
```

`CHOCO_JAR` est un chemin vers une archive jar de choco et `CMD_CLASS` est la classe définissant la ligne de commande. L'activation des assertions java force la vérification automatique de toutes les solutions. La verbosité est augmentée pour suivre les différentes étapes. Ensuite, on spécifie un fichier `properties` qui active la visualisation de la meilleure solution de la manière suivante :

```
tools.solution.report=true
tools.solution.export=true
tools.output.directory=java.io.tmpdir
```

Finalement, on spécifie le chemin de l'unique fichier d'instance à résoudre. La commande génère un fichier `.pdf` affiché en figure B.5 contenant la meilleure solution découverte par le solveur. Les logs reproduits ci-dessous donnent un aperçu de la configuration et de la résolution de l'instance. Le système de logs est intégré dans choco et permet d'afficher des informations très précises sur la recherche. Au contraire, on peut diminuer ou supprimer la verbosité. Par exemple, la verbosité par défaut, c'est-à-dire les messages préfixés par les lettres `s`, `v`, `d` et `c`, respectent le format défini dans la CSP Solver Competion.

```
properties...[load-properties:bp-chart.properties]
cmd...[seed:0][db:null][output:null]
properties...[load-properties:/bin-packing-tut/bin-packing-tut.properties]
cmd...[OK]
=====================================================
Treatment of: N1C3W1_A.BPP
loading...[OK]
preprocessing...[status:SAT][obj:17]
model...dim:[nbv:83][nbc:14][nbconstants:44]
solver...dim:[nbv:83][nbc:30][nbconstants:44]
** CHOCO : Constraint Programming Solver
** CHOCO v2.1.1 (April, 2010), Copyleft (c) 1999-2010
- Search completed
  Minimize: NbNE:16
  Solutions: 1
  Time (ms): 152
  Nodes: 25
  Backtracks: 9
  Restarts: 0
checker...[OK]
s OPTIMUM FOUND
v [100, 45, 3] [100, 45] [96, 54] [94, 51, 5] [90, 59] [88, 62] [87, 62, 1] [85, 65]
```

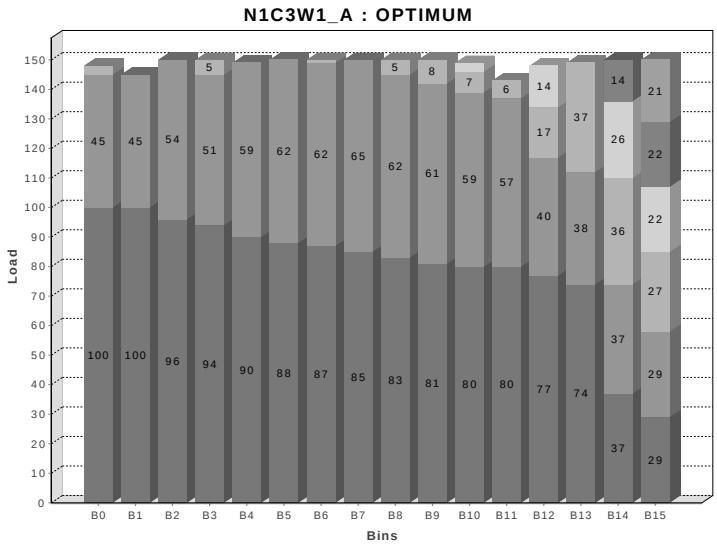


FIGURE B.5 – Une solution optimale de l'instance N1C3W1_A.

```
[83, 62, 5] [81, 61, 8] [80, 59, 7, 3] [80, 57, 6] [77, 40, 17, 14] [74, 38, 37]
[37, 37, 36, 26, 14] [29, 29, 27, 22, 22, 21]
d RUNTIME 0.42
d OBJECTIVE 16
d INITIAL_OBJECTIVE 17
d NBSOLS  1
d NODES 25
d NODES/s 59.38
d BACKTRACKS 9
d BACKTRACKS/s 21.38
d RESTARTS 0
d RESTARTS/s 0
d BEST_LOWER_BOUND 16
d BESTSOLTIME 108
d BESTSOLBACKTRACKS 9
d HEUR_TIME 0
d HEUR_ITERATION 2
c MINIMIZE    N1C3W1_A    0 SEED
c 0.421 TIME    17 PARSTIME    34 PREPROC    78 BUILDPB    140 CONFIG    152 RES

chart...[pdfExport:/tmp/N1C3W1_A.pdf][OK]
cmd...[END]
```

Étape 12 : réaliser des expérimentations.

```
java -cp $CHOCO_JAR $CMD_CLASS -time 60 --export bp-tut-heavy.odb -f instances/
java -cp $CHOCO_JAR $CMD_CLASS --light -time 60 --export bp-tut-light.odb -f instances/
java -cp $CHOCO_JAR $CMD_CLASS --properties bp-cancel-heuristics.properties
     --export bp-tut-heavy-noH.odb -f instances/
```

On lance plusieurs commandes pour comparer des variantes de notre méthode de résolution. Dans la première, on lance la méthode par défaut (modèle *Heavy*) où on (a) impose une limite de temps de 60 secondes, (b) active l'intégration de la base de données en indiquant le nom du fichier `oobase`, et (c) spécifie le répertoire contenant les instances. L'ordre de traitement des fichiers lors de l'exploration récursive du répertoire reste identique d'une exécution à l'autre. Dans la seconde, une option active le modèle *Light*. Dans la troisième, on spécifie un fichier `.properties` qui désactive les heuristiques *first fit* et *best fit* dans le modèle *Heavy* et impose une limite de temps :

```
tools.preprocessing.heuristics=false
tools.cp.model.light=false
```

```
cp.search.limit.type=TIME
cp.search.limit.value=60000
```

Nous résumons brièvement les principaux résultats. Tout d'abord, 24 des 36 instances étudiées sont résolues uniquement grâce à la combinaison des heuristiques et de la borne inférieure. Le modèle *Light* résout seulement 4 instances supplémentaires. Le modèle *Heavy* résout encore 5 autres instances portant le total à 33 instances résolues sur 36. On constate que les performances ne sont presque pas affectées par la suppression de l'heuristique contrairement à ce qui se passait dans le chapitre 6. L'explication est simple, notre stratégie de branchement est directement inspirée de l'heuristique *best fit*. En cas de suppression des heuristiques, la première solution découverte est identique à celle de *best fit*.
La figure B.6 donne un aperçu des rapports générés automatiquement dans le fichier de base de données.

SOLVERS SUMMARY (COMPACT)

Date: 11/02/2011

STATUS OPTIMUM FOUND

NAME	RUNTIME	NB_SOLS	OBJ	TIME	NODES	BACKT.	RESTARTS	VARS	CSTR
N1C1W1_A	0,07	0	25	0	0	0	0	0	0
N1C1W1_B	0,02	0	31	0	0	0	0	0	0
N1C1W2_B	0,01	0	30	0	0	0	0	0	0
N1C1W4_A	0,03	0	35	1	0	0	0	153	68
N1C1W4_B	0,01	0	40	0	0	0	0	0	0
N1C2W1_A	0,00	0	21	0	0	0	0	0	0
N1C2W1_B	0,01	0	26	0	0	0	0	0	0
N1C2W2_A	0,01	0	24	0	0	0	0	0	0
N1C2W2_B	0,01	0	27	0	0	0	0	0	0
N1C2W4_A	0,01	0	29	0	0	0	0	0	0
N1C2W4_B	0,01	0	32	0	0	0	0	0	0
N1C3W1_A	0,05	1	16	42	25	9	0	99	30
N1C3W1_B	0,01	0	16	0	0	0	0	0	0
N1C3W2_A	0,00	0	19	0	0	0	0	0	0
N1C3W2_B	0,05	1	20	15	20	0	0	111	39
N1C3W4_A	0,04	1	21	15	23	0	0	114	38
N1C3W4_B	0,02	1	22	6	19	0	0	117	39
N2C1W1_A	31,40	0	48	31375	29054	62557	0	242	90
N2C1W1_B	0,01	0	49	0	0	0	0	0	0
N2C1W2_A	2,29	0	64	2254	6930	9329	0	290	124
N2C1W2_B	0,01	0	61	0	0	0	0	0	0
N2C1W4_A	0,01	0	73	0	0	0	0	0	0
N2C1W4_B	0,03	0	71	6	5	12	0	311	138
N2C2W1_A	0,01	0	42	0	0	0	0	0	0

FIGURE B.6 – Aperçu d'un rapport généré par oobase.

Annexe C

Problème de collectes et livraisons avec contraintes de chargement bidimensionnelles

Le problème de tournées de véhicules est d'application très générale et se situe au cœur de la problématique très actuelle de la réduction des coûts de la logistique et du transport. La version classique de ce problème est la suivante : à partir d'un centre de distribution, un transporteur doit livrer un ensemble de clients ayant chacun une demande connue. Pour réaliser ces livraisons, le transporteur dispose de plusieurs camions. Plusieurs contraintes lui sont imposées : capacité limitée des camions, fenêtres horaires de livraison, respect des conditions de travail des conducteurs ... Ce problème se pose en logistique industrielle (approvisionnement en matières premières, transport inter-usine, distribution de produits finis). De nombreuses applications existent également dans le domaine des services (ramassage scolaire, collecte des ordures ménagères, transport de personnes handicapées ...).
Nous examinons ici cas particulier de problème de tournées de véhicules dans lequel la demande consiste en un ensemble d'articles rectangulaires Les véhicules ont une surface de chargement rectangulaire et une capacité en poids. Ce problème contient donc une composante de placement (voir section 3.5). Nous présentons un modèle exploitant des techniques d'ordonnancement sous contraintes et une contrainte de non-chevauchement en paires d'objets. Nous montrons aussi comment adapter cette contrainte pour prendre en considération le chargement et le déchargement du véhicule lors des tournées avec collectes et livraisons.
Ce travail est le résultat d'une étude et d'évaluations (non fournies) préliminaires dont nous avons conclu à l'époque qu'il n'était pas possible de résoudre un tel problème avec `choco`*. Par contre, le constat que l'ordonnancement sous contraintes jouait un rôle crucial dans la modélisation des problèmes plus complexes (placement, tournées ...) a aiguillé nos recherches vers d'autres horizons. De nos jours, de tels problèmes représentent encore un réel défi pour* `choco`*, mais maintenant à sa portée grâce aux nouvelles contraintes disponibles (voir section 9.1).*

CIRRELT

Centre interuniversitaire de recherche
sur les réseaux d'entreprise, la logistique et le transport

Interuniversity Research Centre
on Enterprise Networks, Logistics and Transportation

Two-Dimensional Pickup and Delivery Routing Problem with Loading Constraints

Arnaud Malapert
Christelle Guéret
Narendra Jussien
André Langevin
Louis-Martin Rousseau

August 2008

CIRRELT-2008-37

Bureaux de Montréal :
Université de Montréal
C.P. 6128, succ. Centre-ville
Montréal (Québec)
Canada H3C 3J7
Téléphone : 514 343-7575
Télécopie : 514 343-7121

Bureaux de Québec :
Université Laval
Pavillon Palasis-Prince, local 2642
Québec (Québec)
Canada G1K 7P4
Téléphone : 418 656-2073
Télécopie : 418 656-2624

www.cirrelt.ca

Two-Dimensional Pickup and Delivery Routing Problem with Loading Constraints

Arnaud Malapert[1,2,3,*], Christelle Guéret[2], Narendra Jussien[2], André Langevin[1,3], Louis-Martin Rousseau[1,3]

[1] Interuniversity Research Centre on Enterprise Networks, Logistics and Transportation (CIRRELT)

[2] École des Mines de Nantes, 4, rue Alfred Kastler, B.P. 20722, 44307 NANTES Cedex 3, France

[3] Département de mathématiques et génie industriel, École Polytechnique de Montréal, C.P. 6079, succursale Centre-ville, Montréal, Canada H3C 3A7

Abstract. In this paper, a special case of the vehicle routing problem in which the demands consist in a set of rectangular two-dimensional weighted items is considered. The vehicles have a two-dimensional loading surface and a maximum weight capacity. These problems have a routing and a packing component. A framework to handle the loading of a vehicle is proposed. A Constraint Programming loading model based on a scheduling approach is developed. It is also shown that the non-overlapping rectangle constraint can be extended to handle a practical constraint called sequential loading constraint.

Keywords. Vehicle routing problem with loading constraints, packing problem, constraint programming.

Acknowledgements. The corresponding author wish to thank Laboratoire d'informatique de Nantes-Atlantique (LINA), Institut de Recherche en Communications et Cybernétique de Nantes (IRCCyN), École des Mines de Nantes (EMN), Centre interuniversitaire de recherche sur les réseaux d'entreprise, la logistique et le transport (CIRRELT), Laboratoire Quosséça and Nicolas Beldiceanu.

---

* Corresponding author: arnaud.malapert@polymtl.ca

Dépôt légal – Bibliothèque et Archives nationales du Québec,
Bibliothèque et Archives Canada, 2008

1 Problem description

The *Pickup and Delivery Problem with Two-dimensional Loading Constraints* (2L-PDP) is defined[1] by a quintuplet $\mathcal{P} = (G, K, I, M, O)$ where G is an undirected graph, K represents the fleet of vehicles, I the set of customers, M the set of items and O the objective.
Each vehicle $K = \{1, \ldots, m\}$ has a weight capacity P_k and a rectangular loading surface of width W_k and height H_k. The vehicle has a single opening for loading and unloading items. The demand of each client $i \in I = \{1, \ldots, n\}$ consists in a set of m_i items to be carried from a pickup site i^+ to a delivery site i^-. $\mathcal{G}$ is a complete graph $\mathcal{G} = (V_0, E)$ where $V_0 = \cup_{i \in I}\{i^+, i^-\} \cup \{D\}$ and D is the depot. E is the set of edges (i, j) between any pair of vertices, with associated cost c_{ij}. Each item $m_i^j \in M$ is associated to a client i and has a unique index $j \in \{1, \ldots, m_i\}$, a width w_i^j, a height h_i^j and a weight p_i^j. We assume that the set of items demanded by each client can be loaded into a single vehicle.
Split deliveries are not allowed, *i.e.* the entire demand must be loaded on the same vehicle.The aim is to find a partition of the clients into routes of minimal total cost, such that for each route there exists a *feasible loading* of the items into the vehicle loading surface. Of course, this feasible loading must satisfy the capacity constraint.
A feasible loading is defined as follows: first, it must satisfy the packing constraint, items cannot overlap each other and they must fit completely into the vehicle; second, only *orthogonal* packing are allowed, each item has its edges parallel to the sides of the loading surface. Most works on 2L-CVRP (see section 2) consider *oriented* packing, *i.e.*, items cannot be rotated. In comparison to the traditional packing problem, the load must satisfy a practical constraint called *sequential loading* [2]. This constraint requests that, when visiting a client, his items can be unloaded from the vehicle by means of forklift trucks without having to move items belonging to other clients along the route. When this constraint is not required, allowing thus for re-arrangements of the items in the vehicle at the clients' sites, the problem has an *unrestricted loading*.

2 Related works

2.1 Vehicle routing problem with loading constraints

Most of the *Vehicle Routing Problem* (VRP) solving techniques are focused on the route construction and improvement. The capacity constraint is the only constraint related to the load of the vehicle. But many real-world transportation problems have to deal with both routing and packing.
The first transportation problem with loading constraints is the *Pickup and Delivery Traveling Salesman with LIFO Loading* (TSPLL) [3] where a single rear-loaded vehicle must serve a set of customer requests. Consequently, loading and unloading operations must be performed according to a Last-In First-Out

[1] with the formalism introduced by Gendreau and *al.* [1]

(LIFO) policy.
The *Two dimensional Capacited Vehicle Routing Problem with Loading Constraints* (2L-CVRP) was addressed by Iori and *al.* [2], through branch-and-cut and later by [1, 4], through Tabu Search (TS) and Ant Colony Optimization (ACO). Gendreau and *al.* [1] adapt their formulation to solve *Three-dimensional Capacited Vehicle Routing Problem with Loading Constraints* (3L-CVRP) [5].
Moura and *al.* [6] propose also a framework to integrate *Vehicle Routing Problem with Time Windows* (VRPTW) and the *Container Loading Problem* (CLP). They propose two different methods: a sequential method and a hierarchical method. The sequential method relaxes the sequential loading constraint and allows split deliveries. Therefore, routing and loading are planned at the same time. The hierarchical method addresses the original problem and uses the CLP as a sub-problem of the VRPTW. These two methods use a greedy constructive heuristics for CLP. This heuristics allows to consider extra constraints such as cargo stability.

2.2 Two-dimensional Loading Problem

The *Two-dimensonal Loading Problem* (2LP) consists in finding a feasible loading for a given route. It can be seen as a *containment problem* with additional constraints. The containment problem consists in packing a set of items in a single bin of given width and height. Three models are available to encode a solution of a packing problem: a permutation model, a coordinates model and an interval graph model.
Iori and *al.* [4] propose two approaches to solve the loading sub-problem for a given route: local search and truncated branch-and-bound. Both methods use a lower bound. If the lower bound can not prove infeasibility, they try to compute an upper bound. Local search methods are based on two permutation heuristics: *Bottom-Left* and *Touching Perimeter*. For each packing position, the heuristics checks the sequential loading constraint on already packed items. A failure can only be detected when there are no more feasible positions for an item or if the height of the packing is greater than the height of the loading surface.
At each node of the search tree, the truncated branch-and-bound tries to place each remaining item in a limited subset of positions. Backtracking is performed when an item cannot enter any corner point. The procedure is halted when it reaches a maximum number of backtracks, or when a feasible solution is found.

There is no CP methods to solve the loading problem, but some packing methods have been proposed. [7, 8] use the coordinates model whereas Moffit and *al.* [9] uses a meta-CSP based on the graph model. Our model integrates some of their techniques.

3 Loading model

The loading sub-problem is solved very often during search and some instances are infeasible. Our goal is to detect infeasible loading as quickly as possible, to escape from infeasible regions of the search space.
Our model uses lower bounds, redundant constraints issued from the scheduling field (see Section 3.1) and the sequential loading constraint (see Section 3.3).
Scheduling constraints use the notion of *tasks* and *resources* (see section 3.1). Our model uses a pair of tasks to represent the origin and the dimensions of an item, it implies that we use the coordinates model. This representation allows an efficient interaction between packing and scheduling constraints.
The routes are also represented by a set of tasks which allows to add scheduling constraints (precedence, time windows,...) on or between clients. A routing solving technique must propagate its decisions on these tasks. These tasks are an interface to determine constraints on the loading due to the route.

Let i be a client, $T_i^k = (S_i^k, P_i^k)$ denotes a task of starting date S_i^k and processing time P_i^k associated to route R_k. If the route R_k does not serve the client i then $P_i^k = 0$, otherwise T_i^k must satisfy the properties presented in Figure 1. For each item, let defined $T_{is}^X = (X_i^s, w_i^s)$ (resp. $T_{is}^Y = (Y_i^s, h_i^s)$) a task associated to the dimension of m_i^s along X (resp. Y) axis (see Figure 1). Let $\ll$ denote a precedence relation between two nodes of the same route.

3.1 Scheduling constraints

In this section, we present the scheduling notions and constraints used in our model. A resource has a fixed capacity. A task has a requirement for each resource. The *cumulative* constraint enforces that at each point in time, the cumulated height of the set of tasks that overlap that point, does not exceed a given limit. The height of a task is its resource requirement. An *unary resource* executes one activity at a time. A *disjunctive* constraint handles a unary resource.

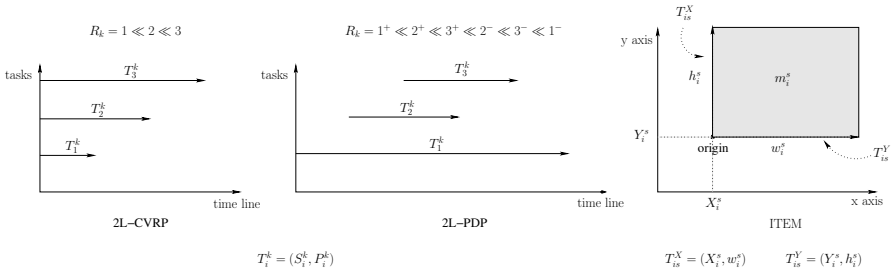


Figure 1: The arrival time of vehicle k at the pickup (resp. delivery) site of client i is the starting (resp. ending) time of T_i^k. The starting time of T_{is}^X (resp. T_{is}^Y) is X_i^s (resp. Y_i^s) and its duration w_i^s (resp. h_i^s), this pair of tasks represents the packing of m_i^s.

A necessary condition for the cumulative constraint is obtained by stating a disjunctive constraint on a subset of tasks T. For each pair of tasks of T, the sum of the two corresponding minimum heights is strictly greater than the capacity. For each cumulative constraints, we add this necessary condition.
Temporal constraints, or precedences, are linear constraints between a pair of tasks. Precedences could come from constraints, decisions and propagation. Indeed, some heuristics use commitment techniques, *i.e.* assign a starting time and duration to a task, others use precedences as decisions. Propagation rules for cumulative and disjunctive could also be modified to deduce precedences.

3.2 Capacity constraints

For each route R_k, the capacity constraint is represented by a cumulative resource of capacity P_k. The requirement of T_i^k is $\sum_{j=0}^{m_i} p_i^j$. In a similar way, we add a cumulative resource of capacity $W_k \times H_k$ in which the requirement of T_i^k is $\sum_{j=0}^{m_i} w_i^j h_i^j$. This redundant constraint triggers a lower bound, proposed by Clautiaux and *al.* [7], which checks packing feasibility.

3.3 Packing constraints

Sequential loading constraint We show that the Sequential Loading Constraint (SLC) is a specialization of the non-overlapping constraint. Let m_i^s and m_j^t be two items of M. The non-overlapping constraint has the following requirements:

$$C^{NO}(m_i^s, m_j^t) \Leftrightarrow$$
$$(X_i^s + w_i^s \leq X_j^{t}) \vee (X_j^t + w_j^t \leq X_i^{s}) \vee (Y_i^s + h_i^s \leq Y_j^{t}) \vee (Y_j^t + h_j^t \leq Y_i^{s})$$

The sequential loading constraint has the following requirements as shown in Figure 2:

$$C^{SLC}(m_i^s, m_j^t) \Leftrightarrow$$
$$\begin{cases} C^{NO}(m_i^s, m_i^t) & \text{if } s \neq t \\ (X_i^s + w_i^s \leq X_j^{t}) \vee (X_j^t + w_j^t \leq X_i^{s}) \vee \left(Y_j^t + h_j^t \leq Y_i^s\right) & \text{if } i^+ \ll j^+ \ll j^- \ll i^- \\ (X_i^s + w_i^s \leq X_j^{t}) \vee (X_j^t + w_j^t \leq X_i^{s}) & \text{if } i^+ \ll j^+ \ll i^- \ll j^- \\ \text{no constraint} & \text{otherwise} \end{cases}$$

It is straightforward to see that C^{SLC} is a restriction of C^{NO}. A global constraint is obtained by considering C^{NO} or C^{SLC} for each pair of M.

Beldiceanu and *al.* [8] describe a generic pruning technique which aggregates several constraints sharing some variables. They specialized this technique to the non-overlapping rectangles constraint. Our purpose is to specialize the algorithm to handle the sequential loading constraint instead of the non-overlapping constraint. We extended the basic filtering algorithm by changing the computation of the *forbidden region*. A forbidden region $R^{no}(m_i^s, m_j^t)$ is a set of origins

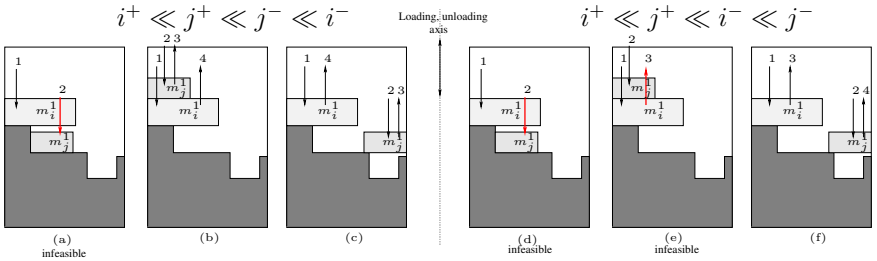


Figure 2: Two clients i_1 and i_2 are on the same route. Each client has a demand which is a single item. We want to load and unload the items with a sequence of four authorized moves along the loading axis.

of m_i^s which overlap with m_j^t. Some work is still needed in order to extend the optimized filtering algorithm. SLC also ensures that each item to be carried by the vehicle k fit completely in its loading surface.

Redundant constraints A way to enhance a CP model is to use redundant constraints, *i.e.*, a constraint which could improve the propagation at each node, although it is already ensured by the model.
A necessary condition for a feasible packing is obtained by stating a cumulative constraint on each dimension. We also can use scheduling techniques extended to packing as energetic reasoning and remaining area estimation [7]. The item's consumption is dependent on the considered node when there is pickup and delivery. If two clients i and j do not share the vehicle, *i.e.* $i^- \ll j^+ \vee j^- \ll i^+$, no constraint links their items.

4 Conclusion and perspectives

In this paper, we have proposed a scheduling based-model with a routing interface and a global constraint which handles SLC for the 2LP. We tried to apply a simple commitment heuristic inspired from the *Bottom-Left* heuristics but it is not efficient. In fact, most packing techniques use reduction procedures and symmetries to reduce the search space but these methods are incompatible with the sequential loading constraint. We want to improve our procedure by postponing the commitment phase. We consider several directions for improving our procedure: (a) searching dimension per dimension; (b) using scheduling branching schemes based on precedences; (c) reducing domains, instead of packing items, to improve the SLC propagation. (a) is equivalent to solving the cumulative problem associated to a dimension before considering the packing. (b) consists in selecting a pair of *critical tasks* and ordering them. The notion of *critical tasks* is based on the resources contention. These two schemes allow for the study of a class of solutions instead of a unique solution. This class of solutions is defined by a configuration on the first dimension or a partial order

between tasks. (c) improves SLC propagation. It ensures that each item has a non-empty forbidden region before it is committed.
Later, we will use this loading model in a constructive routing heuristics and local search methods.

References

[1] Gendreau, M., Iori, M., Laporte, G., Martello, S.: A tabu search heuristic for the vehicle routing problem with two-dimensional loading constraints. Netw. **51**(1) (2008) 4–18

[2] Iori, M., Gonzalez, J.S., Vigo, D.: An exact approach for the vehicle routing problem with two dimensional loading constraints. Transportation Science **41**(2) (May 2007) 253–264

[3] Ladany, S., Mehrez, A.: Optimal routing of a single vehicle with loading constraints. Transportation Planning and Technology **8** (1984) 301–306

[4] Iori, M., Doener, K.F., Hartl, R.F., Fuellerer, G.: Ant colony optimization for the two-dimensional loading vehicle routing problem. Technical report, Università di Bologna (2006)

[5] Gendreau, M., Iori, M., Laporte, G., Martello, S.: A tabu search algorithm for a routing and container loading problem. Transportation Science **40**(3) (August 2006) 342–350

[6] Moura, A., Oliveira, J.F.: An integrated approach to the vehicle routing and container loading problems, Greece, EURO XX - 20th Eur. Conf. on Oper. Res. (2004)

[7] Clautiaux, F., Jouglet, A., Carlier, J., Moukrim, A.: A new constraint programming approach for the orthogonal packing problem. Comput. Oper. Res. **35**(3) (2008) 944–959

[8] Beldiceanu, N., Carlsson, M., Thiel, S.: Sweep synchronization as a global propagation mechanism. Comput. Oper. Res. **33**(10) (2006) 2835–2851

[9] Moffitt, M.D., Pollack, M.E.: Optimal rectangle packing: A Meta-CSP approach. In: Proceedings of the 16th International Conference on Automated Planning & Scheduling (ICAPS-2006). (2006)

Annexe D

Problème d'allocation de ressources et d'ordonnancement pour la maintenance

La gestion efficace de la maintenance des bâtiments peut avoir une influence importante sur le coût total de son cycle de vie, mais aussi sur sa consommation énergétique. Le problème de gestion des bâtiments a été peu étudié jusqu'à présent. Nous présentons une approche en ordonnancement sous contraintes où l'on considère l'allocation des tâches sur un ensemble de ressources. Comme dans le chapitre 7, nous considérons deux modèles, un modèle Light *sans contraintes de partage de ressource, et un modèle* Heavy *basé sur les contraintes* `disjunctive` *et* `useResources` *décrites au chapitre 9. Le modèle* Heavy *est capable de filtrer les domaines des tâches avant que celles-ci ne soient allouées à une ou plusieurs ressources. Les évaluations sur des instances générées aléatoirement montrent que le modèle* Light *est plus rapide sur les petites instances, mais que le modèle* Heavy *passe mieux à l'échelle.*
Ce travail n'est pas intégré dans le corps de ce manuscrit car nous le jugeons inachevé. Des axes de réflexion importants concernent l'amélioration du modèle Light*, la comparaison de différentes stratégies de branchement, et la spécification « définitive » de la contrainte* `useResources`*.*

Modelling a Maintenance Scheduling Problem with Alternative Resources

Aliaa M. Badr[1], Arnaud Malapert[2], and Kenneth N. Brown[1]

[1] Cork Constraint Computation Centre, University College Cork, Ireland
{a.badr,k.brown}@4c.ucc.ie
[2] EMN/LINA UMR CNRS 6241, CIRRELT
arnaud.malapert@emn.fr

Abstract. Effective management of maintenance in buildings can have a significant impact on the total life cycle costs and on the building energy use. Nevertheless, the building maintenance scheduling problem has been infrequently studied. In this paper, we present constraint-based scheduling models for the building maintenance scheduling problem, where each activity has a set of alternative resources. We consider two different models, one using basic constraints, and the other using our new and modified global constraints, which handle alternative disjunctive resources for each activity to allow propagation before activities are assigned to resources. We evaluate these models on randomly generated problems and show that while the basic model is faster on smaller problems, the global constraint model scales better.

Keywords: Building maintenance scheduling, optional activities, alternative resources, disjunctive global constraint

1 Introduction

The operation of large-scale buildings is a significant commercial cost. Suboptimal performance of heating and ventilation systems wastes energy and impacts on the productivity of occupants. Maintenance for large buildings is estimated to consume each year 2% of the total replacement cost of the building [2], with improved maintenance offering up to 20% reductions in the annual energy cost [3]. Effective management of the maintenance process involves solving a scheduling problem with multiple maintenance engineers, interrelated activities, and complex costs on resource use and delays of activities.

Constraint Programming (CP) is one of the most successful techniques for solving scheduling problems with much of its success based on the use of global scheduling constraints, which enable efficient propagation based on commonly occurring problem substructures. In many practical problems, there are multiple discrete resources, and each activity may be executed by any one of a subset of those resources. However, the most successful constraint filtering techniques assume that resources are already assigned to activities before problem solving begins. Some global constraints do model resources where activities are optional,

but these do not take account of the fact that each activity must eventually be scheduled on some resource. In this paper, we consider the problem of building maintenance scheduling. We develop two different models, one using simple constraints, and the other using global constraints representing the alternative resources and optional activities. In the latter model , we use a modified version of the disjunctive global constraint and a new global constraint useResources, which enables modelling resource requirements of activities, i.e., use k of n resources. Finally, we evaluate the two different models and implementations on randomly generated building maintenance scheduling problems. We show that while the basic model is faster on smaller problems, the global constraint model scales better.

The rest of this paper is organized as follows: Section 2 gives description for the application problem, while Section 3 provides the necessary background for the work presented in this paper. Models developed for the building maintenance scheduling problem are discussed in Section 4 and the specification of our global constraints is presented in Section 5. Section 6 provides the experimental evaluation. Finally, Section 7 concludes with a summary and outline for future work.

2 Problem Description

The building maintenance scheduling problem involves a set of maintenance activities, which must be carried out by a set of maintenance engineers with the aim to assign for every activity a start time and an engineer such that all constraints are satisfied and the associated operating costs are minimized. Generally, maintenance activities are either *planned* or *reactive*. Planned maintenance is carried out at predetermined time intervals to prevent degradation or failure of building components and systems, while reactive maintenance is carried out at short notice in response to reported faults, thus making the scheduling problem dynamic. *Emergency* maintenance is a special case of reactive maintenance that causes threat to health and safety and is usually related to a gas (e.g., gas leak), electricity, or water problem.

Typically, a maintenance schedule should satisfy a variety of constraints. For example, some constraints define inter-dependency relations that might occur between activities including *concurrency*, *precedence*, and *non-overlapping*. Concurrency constraints may require a set of activities to be conducted at the same time, while precedence constraints ensure that one set of activities should conclude before another set starts. Non-overlapping constraints restrict a set of activities from overlapping during their execution, to ensure continued safe operation. Additionally, some activities have required skills, and thus can only be carried out by a qualified staff. Furthermore, maintenance involves disruption to the building occupants, and thus activities may be restricted to specified time windows, which might differ over the scheduling horizon. For example, one activity might have the time window (Mon - Wed, 09:00 - 12:00), while another activity could only be executed during one of the following intervals: (Tues, 11:00

- 14:00), or (Thur, 15:00 - 17:00). Hence, activities might have irregular time windows. Finally, engineers execute one activity at a time and have specified skills, limits on their working hours, and windows on their availability, e.g., engineers cannot be assigned activities during their lunch breaks.

An optimized building maintenance schedule is one that satisfies all constraints while minimizing the associated operating costs including maintenance and energy costs. Energy costs are associated with the delayed scheduling of maintenance for components consuming excess amounts of energy if they are faulty, e.g., heaters. Maintenance costs correspond to cost overheads as a result of scheduling activities out of contracted working hours and penalties due to scheduling activities beyond their contracted response time windows. A response time window states when an activity should be carried out according to a service level agreement (SLA). Typically, the SLA defines an expected minimal level of service and may specify financial penalties for the service provider in case the service falls below the minimal level, e.g., failing to address a serious failure in a timely manner. Reactive maintenance may have energy cost associated with it, while planned and reactive maintenance may both incur maintenance costs. We consider scheduling activities within a time-frame of one week.

3 Background

Building maintenance scheduling problems have been infrequently studied in the literature. To the best of our knowledge, this problem is investigated in only one paper [11]. It presents a building maintenance system that uses RFID in maintenance management and includes a scheduling module. This module uses mathematical programming in scheduling planned maintenance activities, while minimizing the total maintenance time. The scheduling module, however, provides schedules for planned maintenance only. Additionally, no details are given regarding problem modelling and handling of different constraints including restrictions of unary resources and temporal constraints.

Constraint-based scheduling models typically involve rich representations of both activities and resources. Activities are usually modelled using three variables representing their start, end, and duration, with $[est, lct]$ representing the time window in which an activity executes, where est and lct are the activity earliest start and latest completion times, respectively. A coherent activity time window is one that satisfies $lct - est \geq$ minimum duration. Disjunctive global constraints are usually used to model unary resources (i.e., with unit capacity) [5]. This constraint ensures that non-interruptible activities executing over a unary resource are sequenced such that they do not overlap at any time interval. Several filtering algorithms are used in the disjunctive constraint including Edge Finding [8], Not-First/Not-Last [18], Detectable Precedence [18], and Overload Checking [19]. We will call these algorithms the *disjunctive resource filtering algorithms*.

The disjunctive resource filtering algorithms assume that all activities to be scheduled on the resource are known in advance, but in many problems each

activity has a choice of resources, and thus the set of activities allocated to each resource is not fixed until those choices are made. One possible way to handle alternatives is by modelling *optional* activities as proposed by Beck and Fox [6]. Optional activities can be classified into two categories: schedule-based, in which an activity may be omitted from the final schedule (e.g. due to different plans being selected), and resource-based, in which the activity is optional for the resource, but must be included on some resource in the final schedule (e.g. courses allocated to a classroom in a timetable). For both categories, we can maintain a binary allocation variable $e_i^r = \{0, 1\}$ for each activity i and resource r. When $e_i^r = 1$, then activity i is allocated to resource r and is said to be *regular* over that resource; when e_i^r is not yet assigned a value, i is *optional* over r; finally, an activity i is *disabled* over r if e_i^r is set to zero. We focus in this paper on resource-based optional activities.

Some researchers have studied scheduling problems with alternative resources. In [9], Focacci *et al.* presented a general job shop scheduling problem with sequence dependent setup times and alternative resources. They use a constraint called *alternative resources* to apply reasoning over activities with alternative resources. A modified version of this constraint is also presented in [19] to solve scheduling problems with alternative resources. Additionally, some disjunctive resource filtering algorithms have been extended to accommodate reasoning over optional activities. In [17], Vilim *et al.* proposed extensions to detectable precedence, not-first/not-last, and overload checking with time complexity $O(nlogn)$. These algorithms use efficient special data structures, called $\Theta - \lambda$ trees that allow "what if" reasoning over different sets of activities. Furthermore, Kuhnert proposed in [12] an extended version of edge finding that handles optional activities with time complexity $O(n^2)$. In these extended versions, an optional activity is disabled over a resource if it causes its overloading, i.e., no feasible sequencing for activities over the resource exists as activities cannot be processed within their time windows. Additionally, the extended reasoning enables regular activities to modify the time windows of other regular and optional activities over the resource. However, the opposite is not true as optional activities could later be disabled during problem solving. These extended algorithms apply to resource-based and schedule-based optional activities.

In [19], the alternative resources constraint considers a set of T activities and R resources, with the following filtering rules:

$$\forall t \in T, \forall r \in R_t : \; S_t^r \; \cap \; S_t \; = \emptyset \Rightarrow R_t := R_t \setminus \{r\}, \tag{1}$$

$$\forall t \in T, \forall r \in R_t : \; S_t^r \; \cap \; S_t \; \neq \emptyset \Rightarrow S_t^r := S_t^r \; \cap \; S_t \tag{2}$$

$$\forall t \in T : \; S_t := S_t \cap (\bigcup_{(r \in R_t)} S_t^r) \tag{3}$$

For every activity t, S_t represents a non-empty finite of potential start times, while R_t is a non-empty finite set of alternative resources. Additionally, for each

activity $t \in T$ and every alternative resource $r \in R_t$, a non empty finite set of alternative start times $S_t^r := S_t$ is defined.

The aforementioned rules are generally triggered whenever S_t or S_t^r is changed. In rule (1), if no possible start times for an activity t are left over a resource r, then r is removed from the set of alternative resources. In (2), modifications applied to the main possible start times are propagated to its alternatives. Finally, rule (3) propagates updates applied over alternative possible start times to the main possible start times whenever one of these alternatives (i.e., S_t^r) is updated. Throughout this paper, we will refer to these rules by the *alternative resources filtering rules* and use ($\Rightarrow$) in their procedural specification, while using ($\rightarrow$) and ($\leftrightarrow$) in logical constraints.

The scheduling problem with alternative resources is solved in [19] using a number of scheduling constraints including the disjunctive constraint, a sweeping algorithm, which performs global overload checking, and the alternative resource constraint. However, the sweeping algorithm is computationally expensive; its time complexity is $O(n^4)$. Additionally, the disjunctive constraint used in [19] and the scheduling constraints used in [9] do not use the extended disjunctive resource filtering algorithms proposed in [17,12]. Furthermore, to the best of our knowledge, no previous work has investigated constraint-based modelling of resource requirements for activities that are allocated a set of resources selected among alternatives.

4 Building Maintenance Scheduling Models

In this section, we discuss the models developed for the building maintenance scheduling problem. We first start with the basic model in Section 4.1, followed by the global constraint model in Section 4.2. In both models, we handle multi-objective optimization through the *weighted objectives* technique. Weighted objectives combines different objective functions into a single one through a linear weighted summation of these objectives. Scalar weights of objectives are specified according to their relative importance [15].

4.1 The Basic Model

This model is based on [4]. The building maintenance scheduling problem consists of a set of activities $T = \{1....n\}$ and a set of engineers $R = \{1....m\}$. For each activity $i \in T$, we define the following variables: $start_i$ is the start time for the activity, with a domain of integers ensuring time windows are obeyed; d_i is the activity duration [3]; end_i is the end time for the activity, with a domain of integers ensuring time windows are obeyed; c_i is the assigned engineer, with a domain of integers, subset from R, ensuring that only engineers with the required skills can be assigned; x_i is the cost resulting from scheduling activity i

[3] Durations of activities are generally considered as variables. In our problem instances, however, we assume activity duration to be a constant integer regardless of which engineer is assigned the activity.

at its chosen start time; s_i is an auxiliary variable linking the start time to an array of costs. Finally, a variable p represents the total cost of the schedule.

Constants included in the model for each activity i are as follows: dt_i is an array of start times such that each value in the domain of $start_i$ appears exactly once in this array; $tcost_i$ is an array of costs, of the same length as dt_i, representing the cost of each start time.

For each activity $i \in T$, we define the following constraint:

$$start_i + d_i = end_i \tag{4}$$

Since the start time windows and costs in building maintenance scheduling are irregular, we use for each activity i the two parallel arrays dt_i and $tcost_i$ such that if $dt_i[index] = t_s$, where t_s is the start time value and $index$ is its position in the array , then the cost of starting activity i at time t_s is $tcost_i[index]$. We represent this using the two element constraints:

$$element(s_i, dt_i, start_i) \tag{5}$$

$$element(s_i, tcost_i, x_i) \tag{6}$$

The $element(I, S, X)$ constraint states that $S[I] = X$. Additionally, for each pair of activities i and j, such that $i \neq j$, we define the disjunctive constraint which ensures that two activities cannot be executed at the same time by the same engineer:

$$c_i = c_j \rightarrow (end_i \leq start_j) \vee (end_j \leq start_i) \tag{7}$$

Depending on the problem instance, we also add constraints of the following types:

$$Concurrency: \ start_i = start_j \tag{8}$$

$$Precedence: \ end_i \leq start_j \tag{9}$$

$$Non-overlapping: \ (end_i \leq start_j) \vee (end_j \leq start_i) \tag{10}$$

$$\forall S, \ AllDifferent(\{c_i : \ i \in S\}) \tag{11}$$

The first three constraint types represent inter-dependency relations that might occur between activities. Additionally, for each maximal set S of concurrent activities, the redundant constraint in (11) is defined to restrict concurrent activities to be assigned to different engineers. We then specify total cost of the schedule as:

$$p = \sum_{i \in T} x_i \tag{12}$$

Finally, the objective function is to minimize p.

4.2 The Global Constraint Model

In addition to the basic model above, we consider another model, which uses global constraints to capture known relationships between activities and resources, in an attempt to improve propagation. Firstly, we add the allocation variables. For each activity $i \in T$ and for every engineer $r \in R$, we define a binary variable e_i^r, to indicate whether the activity i is assigned to engineer r. If engineer r is not qualified for the activity (i.e., is not among the set of alternative engineers with the required skills), e_i^r is set to zero. This requires a channeling constraint (13), which links c_i with the e_i^r:

$$\forall i \in T, \forall r \in R: \; c_i = r \leftrightarrow e_i^r = 1 \tag{13}$$

Secondly, we remove all constraints of type (7), which only propagate once activities have been assigned to engineers, and instead add two global constraints:

$$\forall r \in R: \; AltDisj([start_1, .., start_n], [d_1, .., d_n], [end_1, .., end_n], [e_1^r, .., e_T^r]) \tag{14}$$

$$\forall i \in T : useResources(1, \; start_i, \; d_i, \; end_i, \; [e_i^1, ...e_i^m], \; R) \tag{15}$$

For each engineer, the alternative disjunctive constraint (14) lists all activities which could possibly be executed by that engineer, and ensures that if they are assigned to the engineer then they do not overlap. Additionally, the useResources constraint (15) ensures that each activity must be eventually assigned to exactly one engineer. We use two separate global constraints that reason about alternative resources to accommodate the use of hypothetical domains and enable modelling heterogeneous resource requirements as explained in Section 5.

Finally, we add a cumulative constraint (16) to enable early propagation over the domains of start time variables, where h_i is a unit height for activity i. As little pruning can take place until activities are assigned a resource, it is possible that multiple activities could be assigned the same start time during search even though they are sharing some resources. Hence, the cumulative constraint restricts the number of activities taking the same start time to be at most equal to the total number of engineers.

$$cumulative([start_1, ...start_T], [d_1, ..., d_T], [end_1...., end_T], [h_1, .., h_T], \; m) \tag{16}$$

5 The Global Constraints

In this section, we present our modifications to the alternative disjunctive constraint (the one that includes the extended disjunctive resource filtering algorithms) and introduce our new global constraint useResources. Note that these constraints are not specific to our building maintenance scheduling problem, and can be applied to any scheduling problem involving alternative resources.

5.1 The Alternative Disjunctive Constraint

Scheduling problems with alternative resources can be defined as follows:

Definition 1 *"There is a set of activities* $T = \{1....n\}$ *and a set of resources* $R = \{1.....m\}$. *Every activity* $i \in T$ *has a set of alternative resources* $A \subset R$, *where* $|A| \geqslant 1$. *Furthermore, for every* $i \in T$ *and for every* $r \in R$, *a boolean variable* e_i^r *is defined to reflect the status of activity* i *over resource* r. *We must allocate for every activity* i *a resource* r *such that:* $\forall i, j \in T, \forall r \in R : (e_i^r = 1) \wedge (e_j^r = 1) \rightarrow (end_i \leq start_j) \vee (end_j \leq start_i)$*".*

Alternative resources filtering rules discussed in [19] do not use binary allocation variables to represent the activity status over its alternative resources. These binary variables are commonly used to represent alternatives as shown in Section 3. They are also used in the extended disjunctive resource filtering algorithms proposed in [17,12]. Hence, we modify the alternative resources filtering rules to enable an easy integration with the extended disjunctive resource filtering algorithms and to ensure compatibility with the useResources constraint discussed in Section 5.2.

Considered from the resource point of view, a subset of the activities may be scheduled on the resource. This leads to the following definition for the alternative disjunctive constraint : $AltDisj([start_1, .., start_n], [d_1, .., d_n], [end_1, .., end_n], [e_1^r, .., e_n^r])$. The straightforward specification for the constraint would require, for each possible activity on the resource, a new variable $start_i^r$ representing the possible start times for the activity on that resource. However, this creates two problems. First, if filtering on the resource removes all possible start times for the optional activity, that creates an empty domain, which will cause backtracking, even though no inconsistency has been discovered. Instead, an optional activity should be disabled over the resource, and thus the internal solver behavior should be modified. Additionally, introducing these alternative variables and domains modifies the constraint network, and thus interferes with reformulation methods and search heuristics, particularly those based on degree.

To overcome these drawbacks, we introduce hypothetical domains. A hypothetical domain Dh_i^r represents the time window of an optional activity i over an alternative resource r. Initially, $Dh_i^r = Dm_i = [est_i, lct_i]$. Dm_i represents bounds for activity i main time window, which reflects a unified view for different time windows available over the activity alternative resources. We use bounds representation for hypothetical domains rather than a domain of values as the extended disjunctive resource filtering algorithms apply bound consistency reasoning. This representation also enables reasoning on variable durations.

The use of hypothetical domains limits the number of variables that need to be created. Secondly, when an activity becomes regular over a resource, its hypothetical time window is reflected over the main time window (i.e., $e_i^r = 1 \Rightarrow Dm_i := Dm_i \cap Dh_i^r$) and the alternative disjunctive constraint references the activity main time window instead of its hypothetical one. Hence, propagation on a resource filters the hypothetical domain and transfer to the main domain once the activity becomes regular. Finally, when a hypothetical domain becomes

empty or incoherent, the CP solver sets the corresponding binary allocation variable to zero.

We modify the alternative resources filtering rules to enable using hypothetical domains and binary allocation variables. The first two rules are redefined as follows:

$$\forall i \in T:\ e_i^r = \{0,1\} \wedge\ |Dh_i^r \cap Dm_i| < min(d_i) \Rightarrow e_i^r := 0 \tag{17}$$

$$\forall i \in T:\ e_i^r = \{0,1\} \wedge\ |Dh_i^r \cap Dm_i| > min(d_i) \Rightarrow Dh_i^r := Dh_i^r \cap\ Dm_i \tag{18}$$

These action rules are triggered whenever the domain of one of the activity variables (start, end, or duration) is modified, initialized or values are removed from its domain, and thus the main time window is updated. They are also triggered when a hypothetical domain is updated by the extended disjunctive resource filtering algorithms, where $min(d_i)$ is the minimum duration of activity i. For example, if the domain of an activity related variable is reduced to a singleton, rule (18) updates the hypothetical domain. After that, rule (17) is checked to ensure that the time window for the updated hypothetical domain is still consistent and if this is not the case, then the activity is disabled over its alternative resource. To illustrate, we assume having an activity 1 with variables $start_1 = [5{-}9]$, $end_1 = [7{-}11]$, and $d_1 = 2$. Hence, $Dm_1 = [5{-}11]$. Additionally, we assume that this activity has two alternative resources and its hypothetical domain over the first resource is $Dh_1^1 = [5-11]$ and is $Dh_1^2 = [8-11]$ over the second resource. During problem solving, if $start_1$ is assigned the value 5, then activity 1 is disabled over the alternative resource 2. Note that when values are removed from the domain of an activity related variable, a hypothetical domain is updated only if the removed value contributes to its boundaries. Additionally, the extended disjunctive resource filtering algorithms are triggered as normal when the hypothetical domains and/or main domains are modified.

5.2 Allocation Constraint: useResources

The representation of alternative resources through binary variables mandates the use of constraints to ensure that resource requirements are satisfied. Additionally, an activity might have different time windows over different resources, and thus it is necessary to ensure that their updates are propagated back to the activity main time window. Typically, every resource maintains hypothetical domains corresponding to its optional activities and it does not retain any information related to other hypothetical domains for these activities over other alternative resources. Additionally, the alternative resources filtering rule (3) could be generalized to represent heterogeneous resource requirements such that an activity could be allocated to k resources selected among a given subset of alternatives instead of one resource. Hence, to simplify problem modelling and gain

in flexibility, this rule is defined in a separate new global constraint called useResources. Therefore, an activity could specify different resource requirements like for example a demand for k_1 workers and k_2 machines.

The constraint is defined for an activity i and a requirement $1 \leq k \leq m$ as follows: useResources(k, $start_i$,d_i, end_i, $[e_i^1, ...e_i^m]$, R). This constraint integrates the boolean sum constraint $\sum_{j=1..m} e_i^j = k$ or $\sum_{j=1..m} e_i^j \geq k$ defined over the binary allocation variables. Additionally, it includes the filtering rule that aggregates deductions emerging from alternative time windows until resource requirement is satisfied. Let $\min_P(k, val_i^r)$ be the k-th minimum of val_i^r among a subset P from the set of resources R. For example, the third minimum of the set $\{1, 2, 2, 3\}$ is 2. Rule (19) is the modified version of rule (3) and is applied when a single resource is needed. Rule (20) is used when it is required to allocate k resources.

$$\forall i \in T,\ k = 1 :\ Dm_i := Dm_i \cap \bigcup_{1 \in e_i^r} Dh_i^r \tag{19}$$

$$\forall i \in T,\ k > 1 :\ Dm_i := Dm_i \cap \left[\min_{1 \in e_i^r}(k, est_i^r), \max_{1 \in e_i^r}(k, lct_i^r)\right] \tag{20}$$

In addition, the framework offers a great flexibility since many types of resource requirements could be integrated by defining new constraints interacting with hypothetical domains.

6 Experimental Evaluation

We have developed the presented models in Choco [13]; an open source Java constraint solver, which provides access to recent implementations for a wide range of global constraints. The flexibility of its scheduling component reduces the implementation effort, while providing satisfactory results in solving job-shop and open-shop scheduling problems as shown in the fourth international CSP solver competition [1] and it demonstrates a satisfactory performance with dedicated approaches as shown in [14,10]. The basic model is relatively straightforward, and maps exactly onto Choco primitives. The global constraint model, however, uses two global constraints. The alternative disjunctive constraint is already available in Choco, but without hypothetical domains, the integrated alternative resources filtering rules presented in Section 5.1, and no filtering is carried out, by the extended disjunctive resource filtering algorithms, over the time window of an optional activity until it becomes regular. We implement the modified alternative disjunctive constraint, the useResources constraint, and the extended edge finding filtering algorithm [12].

The extended disjunctive resource filtering algorithms are not idempotent. They are executed in iterations in the alternative disjunctive constraint till no further changes are found by any filtering algorithm, i.e., a fixed point is reached. The order in which these algorithms are called is the one discussed in [14].

This order reduces the total solving time, with no effect on the resulting fixed point, by minimizing the number of sorts required by the data structures. Our experiments show that the extended edge finding algorithm is computationally expensive. Hence we exclude this algorithm in our experimental evaluation.

In our experiments, we consider (M_1) to be the global constraint model described in Section 4.2, while (M_2) is the basic model illustrated in Section 4.1. In both models, we use the variable ordering heuristic ModReg that uses cost based reasoning to guide the search process towards the most viable course [4], based on the Regret heuristic [16]. The regret is the additional cost to be paid for a variable assignment over the cost lower bound if the variable is not assigned the suggested value at its lower bound. The regret heuristic suggests assigning the variable with the highest regret first so as to minimize the risk of paying a higher cost if the best assignment becomes infeasible. In the context of the maintenance scheduling problem, regret is calculated as the difference in cost between the earliest and second earliest start times. Additionally, the definition of the regret heuristic is extended in [4] by breaking ties through selecting the variable corresponding to the activity with the minimum number of time slots with zero cost or with minimum duration. This heuristic gives the best performance in maintenance scheduling in [4]. The branching strategy used in our models assigns for each selected activity, based on ModReg, the start time variable followed by the engineer variable. Thus, domains of decision variables in both models are enumerated such that the engineer with the earliest starting time is assigned first.

We built a generator that randomly generates test cases with different problem sizes and constraint relations, where problem size is the number of activities in each test case. We depend on randomly generated instances as energy information is not considered in real problems where a conventional first come first serve scheduling mechanism is usually followed. However, the data used in the generator depends on the information gathered from discussions with our collaborators [4].

In the random generator, we use a fixed number of engineers (15). The number of alternative engineers for each activity is generated randomly between one and four. Their identifiers are also generated randomly between one and fifteen. Activities participating in a dependency relation and those that are independent have distributions shown in Table 1a in columns (dep) and (indep), respectively. Additionally, the number of activities that could participate in a dependency relation is generated randomly between two and four and the inter-dependency relation type is also generated randomly.

Available time windows for activity execution are generated through two separate operations: the selection of days and selection of time frames within these days. First, we generate randomly for every activity up to two separate ranges of days per week and up to two separate time frames within every generated range of days. For a single range of days, we choose either all five days Mon-Fri

[4] Industrial partner Spokesoft, Civil Engineers, and the Building and Estates office at University College Cork

(with probability 0.6), or a random sub-interval of Mon-Sun (with probability 0.4). If two separate ranges of days are required, then we generate randomly two non-overlapping sub-intervals of Mon-Sun. One time frame is generated with distributions 0.6 for a 09:00 - 17:00 time frame and 0.4 for a time frame that is generated randomly between 09:00 - 20:00. The two separate time frames involve the random generation of two non-overlapping time frames between 09:00 - 20:00.

The activity maintenance type (planned (plan), reactive (reac), and emergency (emerg)) is generated with the distributions presented in Table 1a. The response time window is generated randomly depending on the maintenance type; a four hours response window is assumed for emergency, 1, 2, 3, 7, or 15 days for non-emergency reactive maintenance, and a period of up to 30 days for planned maintenance. Note that, for response time windows greater than one week, we consider only the part of the response time window, which is included in the scheduling horizon. Expected durations for activities are generated with distributions shown in Table 1a, where slot granularity is assumed to be 30 minutes. The excess amount of energy, if any, consumed by the malfunctioning component per unit time is generated with distributions representing no excess energy, low, medium or high energy levels as shown in Table 1b. Finally, missed window refers to the time window already passed from the activity response time before its scheduling. The distribution of activities having missed windows and those that do not are shown in Table 1b. Missed window is used to specify the earliest time at which a penalty applies and its size is generated randomly; for non-emergency reactive maintenance it includes up to 15 days and up to 30 days for planned maintenance. We also consider only the part of the missed window which is included within the scheduling time frame.

We test the performance of the presented models using test cases with different problem sizes, and we generate randomly a set of ten scenarios for each problem size. To avoid extended search time, we set a solving time limit of 10 minutes. Tests were carried out using a 2.40 GHz Intel Core 2 Duo PC with 3.48 GB of memory and running Windows XP. Table 2 illustrates experimental results for test cases that include 20, 50, 70, 100, and 120 activities. Columns Avg.FT, Avg.LT, and Avg.TP give the average times in seconds to find the first solution, the best solution, and prove optimality, respectively. Additionally, column Avg.BT gives the average number of backtracks, while Avg.Nd gives the average number of nodes. This table shows that M_2 is generally much faster in comparison to M_1 (only slower in problems with size 100 for the time required to prove optimality and those with size 120 for the time taken to find the best solution). As expected, the embedded reasoning in the global constraints in M_1 significantly reduces the number of nodes and backtracks.

The branching strategy used in the assignment of the start time and engineer variables enables an efficient use of the global constraints in M_1. This is because after an activity is assigned a start time, the alternative disjunctive constraint disables alternative engineers who cannot carry out the activity with its new time window. To study the impact of this strategy, we experimented with a different

branching strategy that assigns first the start time variables for all activities according to the ModReg heuristic, then assigns engineer variables based on the Dom/WDeg heuristic [7]. Results show that using this strategy, the solver failed to provide any solutions for the majority of the test cases within the specified time limit due to incorrect decisions for the assignment of start times taken at the top of the search tree.

To better understand the effect of the global constraints on the model solving ability, we test the two models, using the same settings, over larger problem instances (150, 200, 220, and 240). Table 3 shows the percentages of test cases solved to optimality and those with solutions found within the aforementioned time limit. This table shows that reasoning incorporated in these constraints significantly improves the model solving ability; it enables M_1 to solve all the test cases and prove solution cost optimality for a few of them (20% of test cases with size 150). However, M_2 shows a degrading performance as the problem size increases. These results demonstrate the effect of problem structure exploitation on solver performance when the maintenance scheduling problem becomes more complex.

Table 1: The probability distribution of features in the test case generator

(a) Dependency, duration, and maintenance type

Feature	Dependency		Duration (Slots)				Maintenance Type		
Value	0: dep	1: indep	1	2	3	4	1: plan	2: reac	3: emerg
Prob.	0.55	0.45	0.50	0.30	0.10	0.10	0.50	0.40	0.10

(b) Energy consumption and missed window

Feature	Energy Consumption				Missed Window	
Value	0:no Energy	1:Low	2:Medium	3:High	0:no missed	1: with missed
Prob.	0.45	0.35	0.15	0.05	0.80	0.20

Table 2: Experimental results for problems that include up to 120 activities in the two CP models

Problem	**Avg.FT**		**Avg.LT**		**Avg.TP**		**Avg.BT**		**Avg.Nd**	
Size	**M_1**	**M_2**	**M_1**	**M_2**	**M_1**	**M_2**	**M_1**	**M_2**	**M_1**	**M_2**
20	0.071	0.027	0.091	0.027	0.119	0.038	**0**	0.2	36.8	36.8
50	0.511	0.110	0.828	0.128	0.879	0.156	**0.2**	6.3	129.1	129.1
70	1.126	0.276	2.004	0.355	2.081	0.415	**0.7**	24.2	184.6	184.6
100	2.649	0.859	103.949	40.828	**135.360**	161.572	**2489.6**	113335	**2650.6**	61194
120	4.019	2.194	**23.284**	43.425	144.436	136.458	**3439.4**	23480.1	**3058.6**	16235.6

Table 3: Percentages of test cases with proven optimal and non-optimal solutions

	optimal (%)		With Sol(%)	
Problem Size	**M_1**	**M_2**	**M_1**	**M_2**
150	20.0	10.0	100.0	80.0
200	0.0	0.0	100.0	70.0
220	0.0	0.0	100.0	30.0
240	0.0	0.0	100.0	20.0

7 Conclusions

We have investigated a building maintenance scheduling problem. We have developed a constraint model that involves basic constraints, and another model that uses global constraints to represent activities with alternative resources with the aim to allow propagation before resources are allocated. We also present our modifications to the alternative disjunctive constraint and introduce our global constraint useResources that enables modelling resource requirements. Experimental evaluation of these models shows that the basic model is faster on smaller problems, but as the problem size increases, the global constraint model scales better, and continues to produce solutions when the simple model does not.

In future work, we will conduct more experiments to precisely evaluate the performance of the modified/new global constraints using benchmark scheduling problems. These constraints constitute an appealing approach to solving a wide range of practical scheduling problems including multi-processor task scheduling. We will also focus on investigating a new three stage search strategy inspired from scheduling and packing problems. The first stage determines the relative ordering between pairs of activities linked by non-overlapping constraints, while the second stage (inspired by ModReg and packing heuristics) restricts start time windows of activities before allocating an engineer. This restriction aims to efficiently utilize the alternative disjunctive and useResources constraints. Finally, the third stage assigns starting times to activities. The existence of a solution is very likely in the third stage, since time windows of activities are tight and consistent. Finally, we consider improving the constraint model by the reformulation and deduction of redundant constraints.

Acknowledgments

This work is part of the ITOBO project, funded by Science Foundation Ireland under grant No.07.SRC.I1170. The authors would like to thank École des Mines de Nantes and École Polytechnique de Montréal; Anika Schumann for her valuable comments on a draft of the paper; and Ena Tobin, the Building and Estates office at University College Cork, and our industrial partner Spokesoft for the information provided about maintenance scheduling.

References

1. Fourth international csp solver competition. http://www.cril.univ-artois.fr/CPAI09/.
2. The whole building design guide. National Institute of Building Sciences, 2005.
3. Energy efficiency in buildings. World Business Council for Sustainable Development, 2008.
4. Aliaa M. Badr and Kenneth N. Brown. Building maintenance scheduling using cost-based reasoning and constraint programming techniques. In *8th European Conference on Product and Process Modelling (ECPPM -10)*, September 2010.
5. Philippe Baptiste and Claude Le Pape. Disjunctive constraints for manufacturing scheduling: Principles and extensions. In *In Proceedings of the Third International Conference on Computer Integrated Manufacturing*, 1995.
6. J. Christopher Beck and Mark S. Fox. Constraint-directed techniques for scheduling alternative activities. *Artif. Intell.*, 121(1-2):211–250, 2000.
7. Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'*, pages 146–150, 2004.
8. J. Carlier and E. Pinson. Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, 78(2):146–161, 1994.
9. W. Nuijten F. Focacci, P. Laborie. Solving scheduling problems with setup times and alternative resources. In *4th International Conference on AI Planning and Scheduling*, pages 92 – 101, 2000.
10. Diarmuid Grimes, Emmanuel Hebrard, and Arnaud Malapert. Closing the open shop: Contradicting conventional wisdom. In *Principles and Practice of Constraint Programming*, pages 400–408, 2009.
11. Chien-Ho Ko. Rfid-based building maintenance system. *Automation in Construction*, 18(3):275 – 284, 2009.
12. Sebastian Kuhnert. Efficient edge-finding on unary resources with optional activities. Proceedings of INAP 2007 and WLP 2007 (2007).
13. F. Laburthe and N. Jussien. Choco constraint programming system. http://www.emn.fr/z-info/choco-solver/tex/choco.pdf, February 2010.
14. Arnaud Malapert, Hadrien Cambazard, Christelle Guéret, Narendra Jussien, André Langevin, and Louis-Martin Rousseau. An optimal constraint programming approach to solve the open-shop problem. *Journal of Computing*, (CIRRELT-2009-25), 2009.
15. R. T. Marler and J. S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, 26:369–395, 2004.
16. Michela Milano. *Constraint and Integer Programming: Toward a Unified Methodology*. 2003.
17. Petr Vilím, Roman Barták, and Ondřej Čepek. Extension of o(n log n) filtering algorithms for the unary resource constraint to optional activities. *Constraints*, 10(4):403–425, 2005.
18. Petr Vilím. O(n log n) filtering algorithms for unary resource constraint. In *Proceedings of CP-AI-OR 2004*, pages 335–347. Springer-Verlag, 2004.
19. Armin Wolf and Hans Schlenker. Realising the alternative resources constraint. In *INAP/WLP*, pages 185–199, 2004.

Bibliographie

[1] Arnaud Malapert, Christelle Guéret, Narendra Jussien, André Langevin et Louis-Martin Rousseau : Two-dimensional pickup and delivery routing problem with loading constraints. Rapport technique CIRRELT-2008-37, Centre Inter-universitaire de Recherche sur les Réseaux d'Entreprise, la Logistique et le Transport, Montréal, Canada, 2008.

[2] Arnaud Malapert, Christelle Guéret, Narendra Jussien, André Langevin et Louis-Martin Rousseau : Two-dimensional pickup and delivery routing problem with loading constraints. *In First CPAIOR Workshop on Bin Packing and Placement Constraints (BPPC'08)*, Paris, France, mai 2008.

[3] Arnaud Malapert, Hadrien Cambazard, Christelle Guéret, Narendra Jussien, André Langevin et Louis-Martin Rousseau : An optimal constraint programming approach to solve the open-shop problem. Rapport technique CIRRELT-2009-25, Centre Inter-universitaire de Recherche sur les Réseaux d'Entreprise, la Logistique et le Transport, Montréal, Canada, 2009.

[4] Arnaud Malapert, Hadrien Cambazard, Christelle Guéret, Narendra Jussien, André Langevin et Louis-Martin Rousseau : An optimal constraint programming approach to solve the open-shop problem. *Journal of Computing*, in press, accepted manuscript, 2011.

[5] Diarmuid Grimes, Emmanuel Hebrard et Arnaud Malapert : Closing the open shop : Contradicting conventional wisdom. *In Principles and Practice of Constraint Programming - CP 2009*, volume 5732 de *Lecture Notes in Computer Science*, pages 400–408. Springer Berlin / Heidelberg, 2009.

[6] Aliaa M. Badr, Arnaud Malapert et Kenneth N. Brown : Modelling a maintenance scheduling problem with alternative resources. *In The 9th International Workshop on Constraint Modelling and Reformulation (CP10)*, St. Andrews, Scotland, september 2010.

[7] Aliaa M. Badr, Arnaud Malapert et Kenneth N. Brown : Modelling a maintenance scheduling problem with alternative resources. Rapport technique 10/3/INFO, École des Mines de Nantes, Nantes, France, 2010.

[8] Arnaud Malapert, Christelle Guéret et Louis-Martin Rousseau : A constraint programming approach for a batch processing problem with non-identical job sizes. Rapport technique 11/6/AUTO, École des Mines de Nantes, Nantes, France, june 2011.

[9] Arnaud Malapert, Christelle Guéret et Louis-Martin Rousseau : A constraint programming approach for a batch processing problem with non-identical job sizes. *European Journal of Operational Research*, in revision, submitted manuscript, april 2011.

[10] Francesca Rossi, Peter van Beek et Toby Walsh : *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[11] Willem Jan van Hoeve : The alldifferent constraint : A survey. *CoRR*, cs.PL/0105015, 2001.

[12] Nicolas Beldiceanu et Sophie Demassey : Global constraint catalog, 2006.

[13] J. Gaschnig : Experimental case studies of backtrack vs. waltz-type vs. new algorithms for satisficing assignment problems. pages 268–277, Toronto, 1978.

[14] M. L. Ginsberg : Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[15] Richard M. Stallman et Gerald J. Sussman : Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

[16] Robert M. Haralick et Gordon L. Elliott : Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980.

[17] Rina Dechter et Itay Meiri : Experimental evaluation of preprocessing techniques in constraint satisfaction problems. *In Proceedings of the 11th international joint conference on Artificial intelligence - Volume 1*, pages 271–277, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[18] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre et Lakhdar Sais : Boosting systematic search by weighting constraints. *In* Ramon López de Mántaras et Lorenza Saitta, éditeurs : *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 146–150. IOS Press, 2004.

[19] Philippe Refalo : Impact-based search strategies for constraint programming. *In* Mark Wallace, éditeur : *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 de *Lecture Notes in Computer Science*, pages 557–571. Springer, 2004.

[20] William D. Harvey et Matthew L. Ginsberg : Limited discrepancy search. *In* Chris S. Mellish, éditeur : *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 607–615, San Mateo, août 20–25 1995. Morgan Kaufmann.

[21] Leon Sterling et Ehud Shapiro : *The Art of Prolog*. The MIT Press, Cambridge, Mass., 1986.

[22] The eclipse constraint programming system. http://eclipseclp.org.

[23] Sicstus prolog. http://www.sics.se/isl/sicstuswww/site/index.html.

[24] Chip v5. http://www.cosytec.com.

[25] IBM : IBM Ilog Cplex CP Optimizer. http://www-01.ibm.com/software/integration/optimization/cplex-cp-optimizer/, 2011.

[26] Gecode : Generic constraint development environment. http://www.gecode.org, 2006.

[27] Mistral. http://www.4c.ucc.ie/~ehebrard/mistral/doxygen/html/main.html.

[28] Choco Team : Choco : an open source java constraint programming library. http://choco.mines-nantes.fr, 2011.

[29] Koalog constraint solver.

[30] The comet programming language and system. http://www.comet-online.org/Welcome.html.

[31] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa et Mutsunori Banbara : Compiling finite linear CSP into SAT. *In Principles and Practice of Constraint Programming - CP 2006*, volume 4204 de *Lecture Notes in Computer Science*, pages 590–603. Springer Berlin / Heidelberg, 2006.

[32] Pierre Lopez et François Roubellat, éditeurs. *Ordonnancement de la production*. Productique. Hermes Science, 2001.

[33] Philippe Baptiste, Claude Le Pape et Wim Nuijten : *Constraint-Based Scheduling : Applying Constraint Programming to Scheduling Problems*. Kluwer, 2001.

[34] A. Lahrichi : Scheduling : the notions of hump, compulsory parts and their use in cumulative problems. *C.R. Acad. Sci.*, 294:209–211, February 1982.

[35] Bernard Roy : *algebre moderne et theorie des graphes (tome 2)*. Dunod, Paris, 1970.

[36] Rina Dechter : Temporal constraint networks. *In Constraint Processing*, pages 333–362. Morgan Kaufmann, San Francisco, 2003.

[37] M. Gondran et M. Minoux : *Graphs and Algorithms*. John Wiley & Sons, New York, 1984.

[38] Philippe Baptiste et Claude Le Pape : Edge-finding constraint propagation algorithms for disjunctive and cumulative scheduling. *In Proceedings of the Fifteenth Workshop of the U.K. Planning Special Interest Group*, 1996.

[39] J. Carlier et E. Pinson : Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, 78:146–161, 1994.

[40] Philippe Torres et Pierre Lopez : On not-first/not-last conditions in disjunctive scheduling. 2000.

[41] Yves Caseau et Francois Laburthe : Improved CLP Scheduling with Task Intervals. *In* Pascal van Hentenryck, éditeur : *Proceedings of the 11th International Conference on Logic Programming, ICLP'94*. The MIT press, 1994.

[42] Y. Caseau et F. Laburthe : Disjunctive scheduling with task intervals. Rapport technique, Laboratoire d'Informatique de I'Ecole Normale Superieure, 1995.

[43] Petr Vilím : O(n log n) filtering algorithms for unary resource constraint. *In* Jean-Charles Régin et Michel Rueher, éditeurs : *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, First International Conference, CPAIOR 2004, Nice, France, April 20-22, 2004, Proceedings*, volume 3011 de *Lecture Notes in Computer Science*, pages 335–347. Springer, 2004.

[44] Petr Vilím, Roman Barták et Ondrej Cepek : Extension of O (n log n) filtering algorithms for the unary resource constraint to optional activities. *Constraints*, 10(4):403–425, 2005.

[45] Pascal Van Hentenryck, Vijay Saraswat et Yves Deville : Design, implementation, and evaluation of the constraint language cc(fd). *The Journal of Logic Programming*, 37(1-3):139–164, 1998.

[46] Armin Wolf : Better propagation for non-preemptive single-resource constraint problems. *In Recent Advances in Constraints*, volume 3419 de *Lecture Notes in Computer Science*, pages 201–215. Springer Berlin / Heidelberg, 2005.

[47] John N. Hooker : *Integrated Methods for Optimization (International Series in Operations Research & Management Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[48] J. R. Jackson : Scheduling a production line to minimize maximum tardiness. Rapport technique, University of California Los Angeles, 1955.

[49] Philippe Baptiste, Claude Le Pape et Laurent Peridy : Global constraints for partial CSPs : A case-study of resource and due date constraints. *In Principles and Practice of Constraint Programming CP98*, volume 1520 de *Lecture Notes in Computer Science*, pages 87–101. Springer Berlin / Heidelberg, 1998.

[50] J. Christopher Beck et Mark S. Fox : Scheduling alternative activities. *In AAAI '99/IAAI '99 : Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial intelligence*, pages 680–687, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence.

[51] Armin Wolf et Hans Schlenker : Realising the alternative resources constraint. *In* Dietmar Seipel, Michael Hanus, Ulrich Geske et Oskar Bartenstein, éditeurs : *INAP/WLP*, volume 3392 de *Lecture Notes in Computer Science*, pages 185–199. Springer, 2004.

[52] Sebastian Kuhnert : Efficient edge-finding on unary resources with optional activities. *In* Dietmar Seipel, Michael Hanus et Armin Wolf, éditeurs : *Applications of Declarative Programming and Knowledge Management (Proceedings of INAP/WLP 2007)*, numéro 5437 *in* LNCS, pages 38–53, Berlin, 2009. Springer.

[53] Roman Barták et Ondrej Cepek : Incremental filtering algorithms for precedence and dependency constraints. *In ICTAI*, pages 416–426. IEEE Computer Society, 2006.

[54] Nicolas Beldiceanu et Mats Carlsson : A new multi-resource cumulatives constraint with negative heights. *In* Pascal Van Hentenryck, éditeur : *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, volume 2470 de *Lecture Notes in Computer Science*, pages 63–79. Springer, 2002.

[55] Yves Caseau et François Laburthe : Cumulative scheduling with task intervals. *In* Michael Maher, éditeur : *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 363–377, Cambridge, septembre 2–6 1996. MIT Press.

[56] Luc Mercier et Pascal Van Hentenryck : Edge finding for cumulative scheduling. *INFORMS Journal on Computing*, 20(1):143–153, 2008.

[57] Petr Vilím : Edge finding filtering algorithm for discrete cumulative resources in $(kn\log(n))$. *In Principles and Practice of Constraint Programming - CP 2009*, volume 5732 de *Lecture Notes in Computer Science*, pages 802–816. Springer Berlin / Heidelberg, 2009.

[58] Philippe Baptiste et Claude Le Pape : Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. *Constraints*, 5(1/2):119–139, 2000.

[59] Bernard Roy et B. Sussman : Les problèmes d'ordonnancement avec contraintes disjonctives. Rapport technique, Technical Report Note DS no 9bis, SEMA, Paris, 1964.

[60] H. Dyckoff, G. Scheitauer et J. Terno : *Cutting and Packing*, pages 393–413. J. Wiley & sons, Chichester, 1997.

[61] Harald Dyckhoff : A typology of cutting and packing problems. *European Journal of Operational Research*, 44(2):145–159, 1990.

[62] Gerhard Wascher, Heike Haussner et Holger Schumann : An improved typology of cutting and packing problems. *European Journal of Operational Research*, 127(3):1109–1130, December 2007.

[63] H. Kellerer, U. Pferschy et D. Pisinger : *Knapsack Problems*. Springer, Berlin, Germany, 2004.

[64] Silvano Martello et Paolo Toth : *Knapsack Problems : Algorithms and Computer Implementations*. Wiley, New York, 1990.

[65] François Vanderbeck : Computational study of a column generation algorithm for bin packing and cutting stock problems. *Mathematical Programming*, 86:565–594, 1999.

[66] Armin Scholl, Robert Klein et Christian Jürgens : Bison : A fast hybrid procedure for exactly solving the one-dimensional bin packing problem. *Computers & OR*, 24(7):627–645, 1997.

[67] Ian P. Gent et Toby Walsh : From approximate to optimal solutions : Constructing pruning and propagation rules. *In IJCAI*, pages 1396–1401, 1997.

[68] Ian P. Gent : Heuristic solution of open bin packing problems. *J. Heuristics*, 3(4):299–304, 1998.

[69] Richard E. Korf : An improved algorithm for optimal bin packing. *In* Georg Gottlob et Toby Walsh, éditeurs : *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pages 1252–1258. Morgan Kaufmann, 2003.

[70] Paul Shaw : A constraint for bin packing. *In* Mark Wallace, éditeur : *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 de *Lecture Notes in Computer Science*, pages 648–662. Springer, 2004.

[71] Alex S. Fukunaga et Richard E. Korf : Bin completion algorithms for multicontainer packing, knapsack, and covering problems. *J. Artif. Intell. Res. (JAIR)*, 28:393–429, 2007.

[72] Pierre Schaus et Yves Deville : A global constraint for bin-packing with precedences : Application to the assembly line balancing problem. 2008.

[73] M. A. Trick : A dynamic programming approach for consistency and propagation for knapsack constraints. *Annals of Operations Research*, 118(1-4):73–84, 2003.

[74] Andrea Lodi, Silvano Martello et Michele Monaci : Two-dimensional packing problems : A survey. *European Journal of Operational Research*, 141:241–252, 2002.

[75] Sándor P. Fekete et Jörg Schepers : A general framework for bounds for higher-dimensional orthogonal packing problems. *CoRR*, cs.DS/0402044, 2004.

[76] Sándor P. Fekete, Jörg Schepers et Jan van der Veen : An exact algorithm for higher-dimensional orthogonal packing. *CoRR*, abs/cs/0604045, 2006.

[77] Michael D. Moffitt et Martha E. Pollack : Optimal rectangle packing : A Meta-CSP approach. *In Proceedings of the 16th International Conference on Automated Planning & Scheduling (ICAPS-2006)*, 2006.

[78] Richard E. Korf : Optimal rectangle packing : Initial results. *In* Enrico Giunchiglia, Nicola Muscettola et Dana S. Nau, éditeurs : *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling (ICAPS 2003), June 9-13, 2003, Trento, Italy*, pages 287–295. AAAI, 2003.

[79] S. Jakobs : On genetic algorithms for the packing of polygons. *European Journal of Operational Research*, 88:165–181, 1996.

[80] Richard E. Korf : Optimal rectangle packing : New results. *In* Shlomo Zilberstein, Jana Koehler et Sven Koenig, éditeurs : *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004), June 3-7 2004, Whistler, British Columbia, Canada*, pages 142–149. AAAI, 2004.

[81] François Clautiaux, Antoine Jouglet, Jacques Carlier et Aziz Moukrim : A new constraint programming approach for the orthogonal packing problem. *Comput. Oper. Res.*, 35(3):944–959, 2008.

[82] Jacques Carlier, François Clautiaux et Aziz Moukrim : New reduction procedures and lower bounds for the two-dimensional bin packing problem with fixed orientation. *Comput. Oper. Res.*, 34(8):2223–2250, 2007.

[83] G. Scheithauer : Equivalence and dominance for problems of optimal packing of rectangles. *Ricerca Operativa*, 27(83):3–34, 1997.

[84] Nicolas Beldiceanu et Mats Carlsson : Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint. *In CP'01 : Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, pages 377–391. Springer-Verlag, 2001.

[85] Nicolas Beldiceanu et E. Contejean : Introducing global constraints in CHIP. *Mathl. Comput. Modelling*, 20(12):97–123, 1994.

[86] Nicolas Beldiceanu, Mats Carlsson et Sven Thiel : Sweep synchronization as a global propagation mechanism. *Comput. Oper. Res.*, 33(10):2835–2851, 2006.

[87] R. Graham, E. Lawler, J. Lenstra et A. Rinnoy Kan : Optimization and approximation in deterministic sequencing and scheduling : A survey. volume 5, pages 287–326. North–Holland, Amsterdam, 1979.

[88] Eugene L. Lawler, Jan Karel Lenstra, Alexander H. G. Rinnooy Kan et David B. Shmoys : Sequencing and scheduling : Algorithms and complexity. *In* S. C. Graves, A. H. G. Rinnooy Kan et P. H. Zipkin, éditeurs : *Handbooks in Operations Research and Management Science : Logistics of Production and Inventory*, volume 4, pages 445–522, Amsterdam-London-New York-Tokyo, 1993. North-Holland Publishing Company.

[89] E. L. Lawler, J. K. Lenstra et A. H. G. Rinnooy Kan : Minimizing Maximum Lateness in a Two-Machine Open Shop. *MATHEMATICS OF OPERATIONS RESEARCH*, 6(1):153–158, 1981.

[90] James O. Achugbue et Francis Y. L. Chin : Scheduling the open shop to minimize mean flow time. *SIAM J. Comput*, 11(4):709–720, 1982.

[91] T. Gonzalez et S. Sahni : Open shop scheduling to minimize finish time. *Journal of the Association for Computing Machinery*, 23(4):665–679, October 1976.

[92] S. M. Johnson : Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, 1(1):61–68, 1954.

[93] J.R. Jackson : An extension of Johnson's results on job-lot scheduling. *Naval Research Logistics Quarterly*, 3(3), 1956.

[94] Teofilo Gonzalez et Sartaj Sahni : Flowshop and jobshop schedules : complexity and approximation. *Oper. Res.*, 26(1):36–52, January 1978.

[95] I. Adiri et N. Aizikowitz : Open shop scheduling problems with dominated machines. *Naval Research Logistic*, 36:273–281, 1989.

[96] Fiala : An algorithm for the open-shop problem. *MOR : Mathematics of Operations Research*, 8:100–109, 1983.

[97] D. de Werra : Almost nonpreemptive schedules. *Annals of Operations Research*, 26:243–256, 1990.

[98] D. de Werra et Ph. Solot : Some graph-theoretical models for scheduling in automated production systems. *Networks*, 23(8):651–660, 1993.

[99] Yookun Cho et Sartaj Sahni : Preemptive Scheduling of Independent Jobs with Release and Due Times on Open, Flow and Job Shops. *OPERATIONS RESEARCH*, 29(3):511–522, 1981.

[100] Sartaj Sahni et Yookun Cho : Complexity of Scheduling Shops with No Wait in Process. *MATHEMATICS OF OPERATIONS RESEARCH*, 4(4):448–457, 1979.

[101] Teofilo Gonzalez : Unit Execution Time Shop Problems. *MATHEMATICS OF OPERATIONS RESEARCH*, 7(1):57–66, 1982.

[102] D. de Werra, J. Blazewicz et W. Kubiak : A preemptive open shop scheduling problem with one resource. *Operations Research Letters*, 10(1):9–15, 1991.

[103] E. Taillard : Benchmarks for basic scheduling problems. *European Journal of Operations Research*, 64:278–285, 1993.

[104] Peter Brucker, Johann Hurink, Bernd Jurisch et Birgit Wöstmann : A branch & bound algorithm for the open-shop problem. *In GO-II Meeting : Proceedings of the second international colloquium on Graphs and optimization*, pages 43–59, Amsterdam, The Netherlands, 1997. Elsevier Science Publishers B. V.

[105] C. Guéret et C. Prins : A new lower bound for the open shop problem. *Annals of Operations Research*, 92:165–183, 1999.

[106] D. P. Williamson, L. A. Hall, J. A. Hoogeveen, C. A. J. Hurkens, J. K. Lenstra, S. V. Sevast'janov et D. B. Shmoys : Short Shop Schedules. *OPERATIONS RESEARCH*, 45(2):288–294, 1997.

[107] Rainer Kolisch : Serial and parallel resource-constrained project scheduling methods revisited : Theory and computation. *European Journal of Operational Research*, 90(2):320–333, April 1996.

[108] Christelle Guéret et Christian Prins : Classical and new heuristics for the open-shop problem. *EJOR (European Journal of Operational Research)*, 107(2):306–314, 1998.

[109] H. Bräsel, T. Tautenhahn et F. Werner : Constructive heuristic algorithms for the open shop problem. *Computing*, 51(2):95–110, 1993.

[110] G. Dewess, E. Knobloch et P. Helbig : Bounds and initial solutions for frame ineration in machine scheduling. *Optimization*, 28(3):339–349, 1994.

[111] Herbert G. Campbell, Richard A. Dudek et Milton L. Smith : A Heuristic Algorithm for the n Job, m Machine Sequencing Problem. *Management Science*, 16(10):B–630–637, 1970.

[112] Christian Prins : Competitive genetic algorithms for the open-shop scheduling problem. *Mathematical methods of operations research*, 52(3):389–411, 2000.

[113] Konstantinos Chatzikokolakis, George Boukeas et Panagiotis Stamatopoulos : Construction and repair : A hybrid approach to search in csps. *SETN*, 3025:2004, 2004.

[114] Christian Blum : Beam-ACO : hybridizing ant colony optimization with beam search : an application to open shop scheduling. *Comput. Oper. Res.*, 32(6):1565–1591, 2005.

[115] D. Y. Sha et Cheng-Yu Hsu : A new particle swarm optimization for the open shop scheduling problem. *Comput. Oper. Res.*, 35(10):3243–3261, 2008.

[116] J-P. Watson et J. C. Beck : A Hybrid Constraint Programming / Local Search Approach to the Job-Shop Scheduling Problem. *In CPAIOR*, pages 263–277, 2008.

[117] Eugeniusz Nowicki et Czeslaw Smutnicki : An advanced tabu search algorithm for the job shop problem. *Journal of Scheduling*, 8:145–159, 2005.

[118] P. Brucker, T. Hilbig et J. Hurink : A branch and bound algorithm for scheduling problems with positive and negative time-lags. Rapport technique, Osnabrueck University, may 1996.

[119] Christelle Guéret, Narendra Jussien et Christian Prins : Using intelligent backtracking to improve branch-and-bound methods : An application to open-shop problems. *European Journal of Operational Research*, 127:344–354, 2000.

[120] Christelle Guéret et Christian Prins : Forbidden intervals for open-shop problems. Rapport technique, École des Mines de Nantes, 1998.

[121] U. Dorndorf, E. Pesch et T. Phan Huy : Solving the open shop scheduling problem. *Journal of Scheduling*, 4:157–174, 2001.

[122] Philippe Laborie : Complete MCS-based search : Application to resource constrained project scheduling. *In* Leslie Pack Kaelbling et Alessandro Saffiotti, éditeurs : *IJCAI*, pages 181–186. Professional Book Center, 2005.

[123] J. Christopher Beck : Solution-Guided Multi-Point Constructive Search for Job Shop Scheduling. *Journal of Artificial Intelligence Research*, 29:49–77, 2007.

[124] P. Brucker, A. Gladky, H. Hoogeveen, M. Koyalyov, C. Potts, T. Tautenham et S. van de Velde : Scheduling a batching machine. *Journal of Scheduling*, 1(1):31–54, june 1998.

[125] S. Webster et K.R. Baker : Scheduling groups of jobs on a single machine. *Operations Research*, 43:692–703, 1995.

[126] Chris N. Potts et Mikhail Y. Kovalyov : Scheduling with batching : A review. *European Journal of Operational Research*, 120(2):228–249, January 2000.

[127] Chung-Yee Lee, Reha Uzsoy et Louis A. Martin-Vega : Efficient algorithms for scheduling semiconductor burn-in operations. *Operations Research*, 40(4):764–775, 1992.

[128] Philippe Baptiste : Batching identical jobs. *Mathematical Methods of Operations Research*, 52:355–367, 2000.

[129] R. Uzsoy : Scheduling a single batch processing machine with non-identical job sizes. *International Journal of Production Research*, 32(7):1615–1635, 1994.

[130] E. Coffman, M. Yannakakis, M. Magazine et C. Santos : Batch sizing and job sequencing on a single machine. *Annals of Operations Research*, 26:135–147, 1990.

[131] C. T. Ng, T. C. E. Cheng et J. J. Yuan : A note on the single machine serial batching scheduling problem to minimize maximum lateness with precedence constraints. *Operations Research Letters*, 30(1):66–68, 2002.

[132] Peter Brucker et Mikhail Y. Kovalyov : Single machine batch scheduling to minimize the weighted number of late jobs. *Mathematical Methods of Operations Research*, 43:1–8, 1996.

[133] Denis Daste, Christelle Gueret et Chams Lahlou : Génération de colonnes pour l'ordonnancement d'une machine à traitement par fournées. *In 7ième conférence internationale de modélisation et simulation MOSIM'08*, volume 3, pages 1783–1790, Paris France, 2008.

[134] Fariborz Jolai Ghazvini et Lionel Dupont : Minimizing mean flow times criteria on a single batch processing machine with non-identical jobs sizes. *International Journal of Production Economics*, 55(3):273–280, 1998.

[135] Sujay Malve et Reha Uzsoy : A genetic algorithm for minimizing maximum lateness on parallel identical batch processing machines with dynamic job arrivals and incompatible job families. *Computers & OR*, 34(10):3016–3028, 2007.

[136] Y. Ikura et M. Gimple : Scheduling algorithms for a single batch processing machine. *Operations Research Letters*, 5:61–65, 1986.

[137] IC. Perez, JW. Fowler et WM. Carlyle : Minimizing total weighted tardiness on a single batch process machine with incompatible job families. *Computers and Operations Research*, 32 (2):327–341, 2000.

[138] CS Wang et Reha Uzsoy : A genetic algorithm to minimize maximum lateness on a batch processing machine. *Computers and Operations Research*, 29(12):1621–1640, 2002.

[139] Reha Uzsoy : Scheduling batch processing machines with incompatible job families. *International Journal of Production Research*, 33(10):2685–2708, 1995.

[140] Sanjay V. Mehta et Reha Uzsoy : Minimizing total tardiness on a batch processing machine with incompatible job families. *IIE Transactions*, 30:165–178, 1998.

[141] L.L. Liu, C.T. Ng et T.C.E. Cheng : Scheduling jobs with agreeable processing times and due dates on a single batch processing machine. *Theoretical Computer Science*, 374(1-3):159–169, 2007.

[142] Meral Azizoglu et Scott Webster : Scheduling a batch processing machine with non-identical job sizes. *International Journal of Production Research*, 38(10):2173–2184, 2000.

[143] Purushothaman Damodaran, Praveen Kumar Manjeshwar et Krishnaswami Srihari : Minimizing makespan on a batch-processing machine with non-identical job sizes using genetic algorithms. *International Journal of Production Economics*, 103(2):882–891, 2006.

[144] Purushothaman Damodaran, Krishnaswami Srihari et Sarah S. Lam : Scheduling a capacitated batch-processing machine to minimize makespan. *Robotics and Computer-Integrated Manufacturing*, 23(2):208–216, 2007.

[145] Meral Azizoglu et Scott Webster : Scheduling a batch processing machine with incompatible job families. *Computers and Industrial Engineering*, 39((3-4)):325–335, April 2001.

[146] Lionel Dupont et Clarisse Dhaenens-Flipo : Minimizing the makespan on a batch machine with non-identical job sizes : An exact procedure. *Computers and Operations Research*, 29:807–819, 2002.

[147] N. Rafiee Parsa, B. Karimi et A. Husseinzadeh Kashan : A branch and price algorithm to minimize makespan on a single batch processing machine with non-identical job sizes. *Computers & Operations Research*, In Press, Accepted Manuscript:–, 2009.

[148] Ali Husseinzadeh Kashan, Behrooz Karimi et S.M.T. Fatemi Ghomi : A note on minimizing makespan on a single batch processing machine with nonidentical job sizes. *Theoretical Computer Science*, 410(27-29):2754–2758, 2009.

[149] M.T. Yazdani Sabouni et F. Jolai : Optimal methods for batch processing problem with makespan and maximum lateness objectives. *Applied Mathematical Modelling*, 34(2):314–324, 2010.

[150] Petr Vilím : *Global Constraints in Scheduling*. Thèse de doctorat, Charles University in Prague, Faculty of Mathematics and Physics, Department of Theoretical Computer Science and Mathematical Logic, KTIML MFF, Universita Karlova, Malostranské náměstí 2/25, 118 00 Praha 1, Czech Republic, August 2007.

[151] Denis Daste, Christelle Gueret et Chams Lahlou : A Branch-and-Price algorithm to minimize the maximum lateness on a batch processing machine. *In Proceedings of the 11th international workshop on Project Management and Scheduling PMS'08*, pages 64–69, IStanbul Turquie, 2008.

[152] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh et P. H. Vance : Branch-and-price : column generation for solving huge integer programs. *Operations Research*, 46:316–329, 1998.

[153] Eugene Lawler : Optimal sequencing of a single machine subject to precedence constraints. *Management science*, 19:544–546, 1973.

[154] Peter Brucker : *Scheduling Algorithms — Third Edition*. Springer-Verlag, Berlin-Göttingen-Heidelberg-New York, 2001.

[155] Filippo Focacci, Andrea Lodi et Michela Milano : Cost-based domain filtering. *In CP '99 : Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming*, pages 189–203, London, UK, 1999. Springer-Verlag.

[156] A. Cesta et A. Oddi : Gaining efficiency and flexibility in the simple temporal problem. *In TIME '96 : Proceedings of the 3rd Workshop on Temporal Representation and Reasoning (TIME'96)*, page 45, Washington, DC, USA, 1996. IEEE Computer Society.

[157] Daniele Frigioni, Tobias Miller, Umberto Nanni et Christos D. Zaroliagis : An experimental study of dynamic algorithms for transitive closure. *ACM Journal of Experimental Algorithms*, 6:9, 2001.

[158] David J. Pearce et Paul H. J. Kelly : A dynamic topological sort algorithm for directed acyclic graphs. *ACM Journal of Experimental Algorithms*, 11:1–7, 2006.

[159] Peter Brucker, Andreas Drexl, Rolf Mohring, Klaus Neumann et Erwin Pesch : Resource-constrained project scheduling : Notation, classification, models, and methods. *European Journal of Operational Research*, 112(1):3–41, January 1999.

[160] Claude Le Pape, Philippe Couronne, Didier Vergamini et Vincent Gosselin : Time-versus-capacity compromises in project scheduling. *In Proceedings of the Thirteenth Workshop of the U.K. Planning Special Interest Group*, 1994.

[161] J. Christopher Beck, Andrew J. Davenport, Edward M. Sitarski et Mark S. Fox : Texture-based heuristics for scheduling revisited. *In AAAI/IAAI*, pages 241–248, 1997.

[162] Luby, Sinclair et Zuckerman : Optimal speedup of las vegas algorithms. *IPL : Information Processing Letters*, 47:173–180, 1993.

[163] Toby Walsh : Search in a small world. *In IJCAI '99 : Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 1172–1177, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[164] Huayue Wu et Peter van Beek : On universal restart strategies for backtracking search. *In Principles and Practice of Constraint Programming CP 2007*, volume 4741/2007 de *Lecture Notes in Computer Science*, pages 681–695. Springer Berlin / Heidelberg, 2007.

[165] Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary et Vincent Vidal : Nogood recording from restarts. *In* Manuela M. Veloso, éditeur : *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 131–136, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[166] G. Katsirelos et F. Bacchus : Unrestricted nogood recording in csp search, 2003.

[167] C. Le Pape : Implementation of Resource Constraints in ILOG SCHEDULE : A Library for the Development of Constraint-Based Scheduling Systems. *Intelligent Systems Engineering*, 3:55–66, 1994.

[168] W. Nuijten : *Time and Resource Constraint Scheduling : A Constraint Satisfaction Approach.* Thèse de doctorat, Eindhoven University of Technology, 1994.

[169] Philippe Laborie : Algorithms for propagating resource constraints in ai planning and scheduling : existing approaches and new results. *Artif. Intell.*, 143(2):151–188, 2003.

[170] Diarmuid Grimes et Emmanuel Hebrard : Job shop scheduling with setup times and maximal time-lags : A simple constraint programming approach. *In* Andrea Lodi, Michela Milano et Paolo Toth, éditeurs : *CPAIOR*, volume 6140 de *Lecture Notes in Computer Science*, pages 147–161. Springer, 2010.

[171] Christian Schulte : *Programming Constraint Services : High-Level Programming of Standard and New Constraint Services*, volume 2302 de *Lecture Notes in Computer Science*. Springer, 2002.

[172] Nicolas Beldiceanu, Emmanuel Poder, Rida Sadek, Mats Carlsson et Charlotte Truchet : A generic geometrical constraint kernel in space and time for handling polymorphic k-dimensional objects. Technical Report T2007-08, Swedish Institute of Computer Science, oct 2007.

[173] Gilles Pesant : A regular language membership constraint for finite sequences of variables. *In In International Conference on Principles and Practice of Constraint Programming (CP'04)*, volume 3258 de *LNCS*, pages 482–495. Springer-Verlag, 2004.

[174] Julien Menana et Sophie Demassey : Sequencing and counting with the multicost-regular constraint. *In In International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'09)*, volume 5547 de *LNCS*, pages 178 – 192. Springer-Verlag, 2009.

[175] Petr Vilím : Max energy filtering algorithm for discrete cumulative resources. *In Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 5547 de *Lecture Notes in Computer Science*, pages 294–308. Springer Berlin / Heidelberg, 2009.

[176] Petr Vilím, Roman Barták et Ondrej Cepek : Unary resource constraint with optional activities. *In* Mark Wallace, éditeur : *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 de *Lecture Notes in Computer Science*, pages 62–76. Springer, 2004.

[177] Paolo Toth et Daniele Vigo : *The Vehicle Routing Problem*. SIAM - Monographs on Discrete Mathematics and Applications, 2001.

[178] Alexis De Clercq, Thierry Petit, Nicolas Beldiceanu et Narendra Jussien : A soft constraint for cumulative problems with over-loads of resource. *In Doctoral Programme of the 16th International Conference on Principles and Practice of Constraint Programming (CP'10)*, pages 49–54, 2010.

[179] Fabien Hermenier : *Gestion dynamique des tâches dans les grappes, une approche à base de machines virtuelles.* Thèse de doctorat, École des Mines de Nantes, 2009.

[180] Hadrien Cambazard et Barry O'Sullivan : Propagating the bin packing constraint using linear programming. *In* David Cohen, éditeur : *Principles and Practice of Constraint Programming – CP 2010*, volume 6308 de *Lecture Notes in Computer Science*, pages 129–136. Springer Berlin / Heidelberg, 2010.

[181] Gérard Casanova : Cours de gestion de projet, 2010.

Techniques d'ordonnancement d'atelier et de fournées basées sur la programmation par contraintes

Arnaud MALAPERT

Mots-clés : optimisation combinatoire, programmation par contraintes, ordonnancement, problèmes d'atelier, problèmes de fournées.

Résoudre un problème d'ordonnancement consiste à organiser un ensemble de tâches, c'est-à-dire déterminer leurs dates de début et de fin et leur attribuer des ressources en respectant certaines contraintes. Dans cette thèse, nous proposons de nouvelles approches exactes basées sur la programmation par contraintes pour deux classes de problèmes d'ordonnancement NP-difficiles validées expérimentalement par l'implémentation d'un ensemble de nouvelles fonctionnalités dans le solveur de contraintes choco.

Dans un problème d'atelier, n lots sont constitués chacun de m tâches à exécuter sur m machines distinctes. Chaque machine ne peut exécuter qu'une tâche à la fois. La nature des contraintes liant les tâches d'un même lot peut varier (séquencement global ou par lot, pas de séquencement). Le critère d'optimalité étudié est la minimisation du délai total. Nous proposons d'abord une étude et une classification des différents modèles et algorithmes de résolution. Ensuite, nous introduisons une nouvelle approche flexible pour ces problèmes classiques.

Une machine à traitement par fournées peut traiter plusieurs tâches en une seule opération, une fournée. Les dates de début et de fin des tâches d'une même fournée sont identiques. Le problème étudié consiste à minimiser le retard algébrique maximal de n tâches de différentes tailles sur une machine de capacité b. Conjointement, la somme des tailles des tâches d'une fournée ne doit pas excéder la capacité b. Nous proposons, dans ce contexte, un modèle basé sur une décomposition du problème. Nous définissons ensuite une nouvelle contrainte pour l'optimisation basée sur une relaxation du problème qui améliore sa résolution.

Shop and batch scheduling with constraints

Arnaud MALAPERT

Keywords: combinatorial optimization, constraint programming, scheduling, shop scheduling, batch scheduling.

Solving a scheduling problem consists of organizing a set of tasks, that is assigning their starting and ending times and allocating resources such that all constraints are satisfied. In this thesis, we propose new constraint programming approaches for two categories of NP-hard scheduling problems which are validated experimentally by the implementation of a set of new features within the constraint solver choco.

In shop scheduling, a set of n jobs, consisting each of m tasks, must be processed on m distinct machines. A machine can process only one task at a time. The processing orders of tasks which belong to a job can vary (global order, order per job, no order). We consider the construction of non-preemptive schedules of minimal makespan. We first propose a study and a classification of different constraint models and search algorithms. Then, we introduce a new flexible approach for these classical problems.

A batch processing machine can process several jobs simultaneously as a batch. The starting and ending times of a task are the ones of the batch to which they belong. The studied problem consists of minimizing the maximal lateness for a batch processing machine on which a finite number of tasks of non-identical sizes must be scheduled. The sum of the sizes of the jobs that are in a batch should not exceed the capacity b of the machine. We propose, within this context, a constraint model based on a decomposition of the problem. Then, we define a new optimization constraint based on the resolution of a relaxed problem enhanced by cost-based domain filtering techniques which improves the resolution.