ACADÉMIE D'AIX-MARSEILLE
UNIVERSITÉ D'AVIGNON ET DES PAYS DE VAUCLUSE

Laboratoire d'Informatique d'Avignon (EA 931 - CNRS FRE 2487)
Équipe Recherche Opérationnelle et Optimisation

THÈSE

présentée à l'Université d'Avignon et des Pays du Vaucluse
pour obtenir le diplôme de DOCTORAT

Spécialité : Informatique
École doctorale : Sciences et Agronomie (ED 380)

Intégration des techniques de recherche locale à la programmation linéaire en nombres entiers

par

Émilie DANNA

Soutenue le premier juin 2004 devant le jury composé de :

M. Claude Le Pape, ILOG	Directeur
M. Andrea Lodi, Università di Bologna - D.E.I.S. (Italie)	Examinateur
M. Philippe Michelon, Université d'Avignon	Directeur
M. Michel Minoux, Université de Paris-6	Rapporteur
M. Benoît Rottembourg, Bouygues eLab	Président
M. Éric Taillard, University of Applied Sciences of Western Switzerland (Suisse)	Rapporteur

Table des matières

Remerciements

Je voudrais avant tout remercier Claude Le Pape de m'avoir encadrée, étant toujours disponible lorsque je rencontrais des difficultés et me laissant acquérir plus d'autonomie au fur et à mesure de ma progression. Son expérience, sa vision synthétique, sa rigueur et sa bonne humeur ont été un exemple et un guide précieux pendant ces trois années.

Je voudrais également exprimer toute ma gratitude envers Ed Rothberg auprès duquel j'ai énormément appris. Notre collaboration est un des meilleurs souvenirs que je garderai de mon passage à ILOG.

Je remercie Michel Minoux d'avoir accepté d'être rapporteur de cette thèse et, avant cela, de m'avoir donné goût à la programmation linéaire en nombres entiers pendant mon stage de DEA. Je remercie Éric Taillard pour avoir bien voulu être le deuxième rapporteur de cette thèse et pour les nombreuses améliorations du manuscrit qu'il a proposées. Je remercie enfin Andrea Lodi et Benoît Rottembourg pour l'intérêt qu'ils ont porté à mes travaux en me faisant l'honneur de faire partie de ce jury de thèse.

Je remercie tous les membres du département R&D Optimisation d'ILOG, en particulier Laurent Perron, Philippe Réfalo et Jean-Charles Régin, avec qui j'ai eu plaisir à travailler et qui, chacun de manière différente, ont contribué à me faire progresser.

Je remercie également Philippe Michelon et toute l'équipe Recherche Opérationnelle et Optimisation du Laboratoire d'Informatique d'Avignon pour m'avoir permis de mener cette thèse à bien malgré le peu de temps que j'ai passé à Avignon.

Je remercie enfin tous ceux qui ont pris le temps de relire une partie de cette thèse ou un des articles qu'elle reprend, notamment Christian Artigues, Philippe Baptiste, Dominique Feillet, Philippe Laborie et Francis Sourd — en plus des personnes déjà citées. Leurs commentaires m'ont permis d'améliorer la justesse et la clarté de cet exposé, mais les imperfections qui y subsistent sans doute ne sauraient leur être imputées.

Introduction

Lorsque qu'un problème de programmation linéaire en nombres entiers (PLNE) est résolu par *branch-and-cut*, deux problèmes sont à résoudre simultanément : d'une part, augmenter progressivement la borne inférieure sur la fonction objectif (en cas de minimisation) pour obtenir *in fine* une preuve d'optimalité, d'autre part trouver des solutions entières de coût décroissant pour se rapprocher d'une solution réalisable optimale. De nombreuses familles de coupes ont été décrites dans la littérature pour résoudre le premier problème. Nous nous intéressons ici au second problème qui a été moins étudié.

Les solutions entières sont théoriquement obtenues par simple exploration arborescente, par exemple avec le schéma de branchement suivant qui est le plus couramment utilisé. À chaque nœud, s'il existe au moins une variable x qui prend une valeur fractionnaire f dans la relaxation continue, alors on branche sur cette variable, en imposant sur une branche $x \leq \lfloor f \rfloor$, et sur l'autre branche $x \geq \lceil f \rceil$, ce qui revient dans le cas particulier d'une variable binaire à fixer x respectivement à 0 et 1. Ainsi, lorsque la profondeur dans l'arbre augmente, on s'attend à ce que le nombre de variables fractionnaires diminue et on espère que la relaxation deviendra rapidement entière. Cependant, il est parfois difficile de trouver les nœuds où la relaxation est entière. C'est pourquoi, en pratique, des heuristiques sont mises en œuvre pour obtenir des solutions entières par d'autres moyens que la pure recherche arborescente, notamment grâce à des stratégies d'arrondis successifs de la solution de la relaxation continue au nœud courant ou de réparation de solutions entières irréalisables. Ces heuristiques sont appelées régulièrement au cours de l'exploration de l'arbre de *branch-and-cut*, mais en marge de celui-ci, de façon à ce que les décisions heuristiques prises par ces algorithmes ne perturbent pas l'exploration de l'arbre de recherche et que la possibilité de prouver l'optimum soit ainsi conservée. Ces heuristiques sont utiles en pratique parce qu'elles complémentent efficacement les schémas de branchement. Cependant, il existe des problèmes difficiles sur lesquels les heuristiques existantes pour le *branch-and-cut* ne réussissent pas à trouver des solutions entières de qualité.

Le premier objectif de cette thèse est donc de proposer des heuristiques de programmation linéaire en nombres entiers plus puissantes. Pour atteindre cet objectif, nous nous proposons de considérer les heuristiques pour la programmation linéaire en nombres entiers non comme une technique de pure PLNE mais plutôt comme une coopération entre PLNE et techniques de recherche locale. En effet, prenons l'exemple

des heuristiques qui construisent une solution entière réalisable en arrondissant successivement les variables fractionnaires dans la relaxation continue au nœud courant [18]. Ces heuristiques d'arrondis peuvent être vues comme des algorithmes de recherche locale qui explorent le voisinage de la solution de la relaxation. Il est donc intéressant d'exploiter pour les heuristiques de programmation linéaire en nombres entiers les stratégies bien connues en recherche locale concernant notamment la définition des voisinages, l'intensification et la diversification.

Ce point de vue permet de considérer les heuristiques de PLNE comme un cas particulier des algorithmes hybrides, c'est-à-dire des algorithmes qui mettent en œuvre au moins deux des techniques suivantes : algorithmes spécifiques de recherche opérationnelle, programmation linéaire en nombres entiers, programmation par contraintes, recherche locale. Chacune de ces techniques est performante dans des situations différentes ce qui permet idéalement aux algorithmes hybrides d'être efficaces et robustes vis-à-vis de la taille des problèmes considérés, de leurs caractéristiques numériques, et de la possibilité d'ajouter des contraintes additionnelles. De nombreux problèmes difficiles d'optimisation, en particulier des problèmes industriels complexes, ont été résolus récemment avec succès par des algorithmes hybrides. Nous nous proposons donc d'appliquer les idées mises en œuvre par la communauté des algorithmes hybrides au cas particulier des heuristiques pour la PLNE. Cependant, la coopération entre différentes techniques d'optimisation constitue un domaine nouveau, encore très expérimental et qui manque d'outils conceptuels. La plupart des algorithmes hybrides existants s'appliquent uniquement à un problème précis et souffrent d'une complexité logicielle accrue par rapport aux algorithmes purs, ce qui rend plus complexe leur développement, leur déploiement et leur généralisation. Le deuxième objectif de cette thèse est donc de proposer un paradigme général pour de meilleurs algorithmes hybrides à partir de notre étude du cas particulier des heuristiques pour la PLNE. Nous mettrons en particulier l'accent sur le caractère générique des algorithmes, c'est-à-dire leur capacité à s'appliquer à tout type de problème modélisable dans un système donné, sans information extérieure sur la structure du problème à traiter.

Nous explicitons le cheminement qui nous a permis d'identifier ces deux objectifs dans le Chapitre 1. Nous mettons d'abord en évidence l'intérêt et les insuffisances des algorithmes hybrides existants à travers deux schémas de coopération, la *recherche multiple* et la *décomposition* [40], qui nous permettent d'unifier et de classifier la plupart des algorithmes hybrides décrits dans la littérature. Nous nous intéressons ensuite de manière plus détaillée aux algorithmes hybrides où interviennent des techniques de recherche locale. Nous présentons enfin un état de l'art critique des heuristiques existantes pour le *branch-and-cut*.

Dans le Chapitre 2, nous présentons un schéma de coopération général entre recherche locale et génération de colonnes qui généralise au *branch-and-price* la notion d'heuristique pour le *branch-and-cut*. Dans ce schéma de coopération, la recherche locale sert non seulement à produire rapidement des solutions entières de qualité comme

dans les heuristiques employées en *branch-and-cut*, mais elle introduit également de nouvelles colonnes, ce qui augmente la diversité de l'ensemble des colonnes générées. De plus, ce schéma permet d'intégrer n'importe quel algorithme de recherche locale, en particulier des heuristiques efficaces et spécifiques au problème à résoudre, contrairement aux accélérations existantes du *branch-and-price* qui ne mettent en œuvre que des techniques simples d'arrondis [49]. Nous montrerons sur le problème de routage de véhicules avec fenêtres de temps (jeu de données de Solomon [136]) que ce schéma de coopération permet uniformément d'obtenir des bonnes solutions réalisables plus rapidement que le *branch-and-price* pur tout en conservant la capacité du *branch-and-price* à produire de bonnes bornes inférieures sur la fonction objectif. Cependant, ce schéma de coopération présente deux inconvénients. Premièrement, bien qu'il puisse être appliqué à n'importe quel modèle de *branch-and-price*, il n'est pas générique : un nouvel algorithme de recherche locale spécifique au problème étudié doit être mis en œuvre chaque fois qu'un nouveau modèle de *branch-and-price* est à résoudre. Deuxièmement, ce schéma de coopération obéit encore au paradigme des algorithmes hybrides de recherche multiple [40] dont les défauts sont mis en évidence au Chapitre 1.

Dans le Chapitre 3, nous présentons une heuristique pour le branch-and-cut (RINS : *relaxation induced neighborhood search*) qui ne présente aucun de ces deux inconvénients. Premièrement, c'est un algorithme générique : il peut être appliqué à n'importe quel modèle de programmation linéaire en nombres entiers, sans nécessiter d'autre information que le modèle lui-même. Deuxièmement, c'est un algorithme *conceptuellement hybride*. Ce nouveau paradigme définit des algorithmes qui ne mettent en œuvre qu'une seule technique de résolution mais exploitent et transposent dans ce cadre les concepts de plusieurs techniques. Contrairement aux algorithmes combinant plusieurs composants logiciels, ce paradigme présente le triple avantage de ne pas augmenter la complexité logicielle de l'algorithme hybride par rapport à un algorithme pur, de limiter le risque de spécialisation de l'algorithme hybride à un problème particulier, et de faire facilement bénéficier l'algorithme hybride des progrès réalisés pour la technique d'optimisation sous-jacente. RINS repose sur la technique de *branch-and-cut* et exploite les trois concepts fondamentaux de la recherche locale : la diversification, l'intensification et surtout le concept de voisinage. Nous montrons avec de nombreuses expériences numériques sur un jeu de modèles variés et difficiles que RINS est plus performant que la stratégie par défaut de CPLEX, que le *local branching*, une heuristique de programmation linéaire en nombres entiers développée récemment par Fischetti et Lodi [57], et que *guided dives*, une stratégie d'exploration arborescente développée par Edward Rothberg dont nous présentons également l'algorithme.

Dans le Chapitre 4, nous nous intéressons enfin plus en détail au problème d'ordonnancement d'atelier avec coûts d'avance et retard, problème peu étudié et mal résolu jusqu'à présent, sur lequel RINS et *guided dives* sont particulièrement performants. Nous proposons dans ce chapitre plusieurs améliorations et extensions du modèle disjonctif utilisé notamment par Applegate et Cook [7]. Nous présentons en par-

ticulier plusieurs manières de calculer les coefficients de type *big-M* intervenant dans les contraintes disjonctives, dont l'une est heuristique (MCORE : *big-M COefficient REduction*). Cette heuristique de programmation linéaire en nombres entiers est particulièrement adaptée à ce cas particulier mais nous proposons une généralisation qui pourrait être appliquée à n'importe quel modèle avec des coefficients de type *big-M*. MCORE est aussi un algorithme conceptuellement hybride. Nos résultats sur le jeu de données MaScLib [109] sont compétitifs vis-à-vis des approches existantes. Par ailleurs, ils montrent d'une part qu'une bonne modélisation est complémentaire d'une bonne technique de résolution, d'autre part que des techniques spécifiques au problème étudié sont complémentaires de techniques génériques robustes.

Chapitre 1

Algorithmes hybrides et heuristiques pour le *branch-and-cut* : état de l'art et objectifs

La littérature récente recense de nombreux exemples d'algorithmes hybrides qui obtiennent de bons résultats sur un large éventail de problèmes. Cependant, la coopération entre algorithmes constitue un domaine nouveau et encore très expérimental. Dans ce chapitre, nous présentons d'une part un état de l'art des algorithmes hybrides existants, en particulier ceux où interviennent des techniques de recherche locale, avec pour but de dégager des concepts qui permettent d'unifier et de classifier les travaux présentés dans la littérature. D'autre part, nous donnons un état de l'art des heuristiques pour le *branch-and-cut*. En effet, ces heuristiques correspondent à notre définition des algorithmes hybrides exposée dans la Section 1.1.1, mais la plupart d'entre elles ont été développées dans la communauté de la programmation linéaire en nombres entiers, donc ne correspondent pas à la même problématique que celle de la communauté des algorithmes hybrides. Dans la Section 1.4, nous exposons enfin les deux objectifs de cette thèse qui sont motivés par cette revue critique des algorithmes existants.

Les idées exposées dans ce chapitre sont nées du travail que nous avons effectué avec Claude Le Pape au sein de l'équipe « Solveurs Coopératifs » à ILOG. La première section en particulier est fondée sur un chapitre de livre que nous avons écrit ensemble [40].

1.1 Les algorithmes hybrides

1.1.1 Définition

Dans toute la suite de cet exposé, un algorithme sera dit *hybride* s'il met en œuvre au moins deux des techniques suivantes : algorithmes spécifiques de recherche opérationnelle, programmation linéaire en nombres entiers, programmation par contraintes et recherche locale. À l'opposé, nous désignerons par *pur* un algorithme qui fait intervenir une seule de ces quatre techniques. La coopération entre les différentes composantes d'un algorithme hybride, en d'autres termes comment le travail est partagé et quelles informations sont échangées entre ces composantes, peut se faire selon des modalités très variées. Un exemple très simple de coopération de haut niveau est d'appeler chaque composante successivement. Un autre exemple est de décomposer le problème à traiter en plusieurs sous-problèmes qui seront résolus chacun par une technique différente. La coopération entre les différentes techniques mises en œuvre peut également être plus étroite si des informations plus détaillées sont échangées entre chacune des composantes. Dans la Section 1.1.4, nous décrirons en détail deux schémas de coopération qui englobent la plupart des algorithmes hybrides efficaces et robustes exposés dans la littérature récente.

1.1.2 Intérêt

L'intérêt des algorithmes hybrides est de combiner des techniques qui sont chacune performantes dans des situations différentes, comme nous le détaillerons ci-après. La motivation pour mettre en œuvre un algorithme hybride plutôt qu'un algorithme pur est donc double. D'une part, un algorithme hybride est conçu pour être *efficace*, c'est-à-dire capable de fournir de bonnes solutions en un temps raisonnable et avec des ressources limitées (CPU, mémoire). D'autre part, un algorithme hybride est conçu pour être *robuste*, c'est-à-dire capable de traiter de manière satisfaisante une plus grande variété de problèmes qu'un algorithme pur, approchant (ou améliorant) sur chaque instance les meilleurs résultats donnés par chacune de ses composantes. Un algorithme hybride a pour but d'être robuste vis-à-vis de la taille des problèmes considérés, de leurs caractéristiques numériques, et de la possibilité d'ajouter des contraintes additionnelles afin de traiter des problèmes industriels modélisés de la manière la plus réaliste possible.

Le but d'un algorithme hybride est donc d'offrir un compromis entre efficacité et robustesse plus satisfaisant que celui offert par chacune de ses composantes. En effet, les quatre techniques pouvant intervenir dans un algorithme hybride sont en général soit efficaces sur un ensemble limité de problèmes, soit capables de traiter des problèmes plus variés mais avec des résultats de qualité disparate.

Les algorithmes spécifiques de recherche opérationnelle exploitent la structure de problèmes bien définis pour les résoudre optimalement en un temps borné par une fon-

tion polynomiale de la taille du problème. Si le degré de ce polynôme est faible, l'algorithme polynomial est efficace et robuste vis-à-vis des changements de taille et de caractéristiques numériques des instances. Nous incluons également dans cette classe les algorithmes dont la complexité est une fonction polynomiale des données numériques du problème. Ces algorithmes pseudo-polynomiaux offrent moins de garanties mais sont en général relativement efficaces et robustes. Le fonctionnement de ces deux types d'algorithmes repose en général sur l'application de règles de dominance qui permettent de caractériser les solutions optimales et d'éliminer de la recherche des solutions sub-optimales. Dans la plupart des cas, ces algorithmes sont difficiles à adapter pour prendre en compte des contraintes additionnelles qui modifient la structure du problème et la rendent plus complexe et plus difficile à exploiter. En général, les règles de dominance ne sont plus correctes sur le problème modifié et leur application peut conduire à l'élimination de la solution optimale ou même de l'ensemble des solutions admissibles.

La programmation linéaire en nombres entiers (PLNE) permet de modéliser une plus grande variété de problèmes. Tout problème dont l'objectif et les contraintes sont modélisés par des fonctions linéaires et avec des variables entières peut être résolu par *branch-and-cut*. De nombreux progrès ont été réalisés pendant les quinze dernières années pour accroître l'efficacité et la robustesse des moteurs de résolution de programmation linéaire et de programmation linéaire en nombres entiers vis-à-vis de la taille et des caractéristiques numériques des problèmes considérés. Le *branch-and-cut* permet ainsi de résoudre de manière particulièrement efficace les modèles dont la relaxation continue est une bonne approximation de l'enveloppe convexe des solutions entières, du moins autour de la solution optimale, ou pour lesquels cette relaxation peut être renforcée en ajoutant des coupes. Cependant, il n'est pas facile de formuler un problème quelconque et d'obtenir un modèle qui exhibe l'une ou l'autre de ces propriétés sans augmenter exponentiellement le nombre de variables entières ou le nombre de contraintes nécessaires. De plus, si certaines expressions non-linéaires peuvent être linéarisées, par exemple les fonctions linéaires par morceaux et les disjonctions, ou plus généralement les contraintes logiques, leur modélisation aboutit souvent à des relaxations continues faibles donc à une résolution peu efficace.

Contrairement aux algorithmes spécifiques de recherche opérationnelle et au *branch-and-cut*, la force de la programmation par contraintes réside dans sa capacité intrinsèque à intégrer les contraintes additionnelles. D'une part, l'expressivité de la programmation par contraintes permet de modéliser avec un langage de haut niveau un grand nombre de contraintes. D'autre part, en programmation par contraintes, la résolution repose sur l'enchaînement de réductions de l'espace de recherche déduites à partir de chaque contrainte. En effet, à chaque contrainte est associé un algorithme de filtrage : les domaines des variables apparaissant dans cette contrainte sont réduits en éliminant les valeurs incompatibles avec les décisions prises précédemment (par exemple à un nœud donné de l'arbre de recherche). Le mécanisme de propagation consiste

à examiner à nouveau les contraintes impliquant une variable dont le domaine vient d'être modifié, afin de continuer à réduire les domaines des autres variables. Ainsi, en programmation par contraintes, toute contrainte additionnelle peut être propagée et participer activement à la résolution du problème. Cependant, pour être efficace, la programmation par contraintes nécessite d'une part des algorithmes de filtrage rapides et qui réduisent significativement l'espace de recherche — du moins pour les contraintes critiques du problème. D'autre part, la programmation par contraintes aura plus de chances d'être efficace si l'on dispose d'une borne serrée sur l'objectif à optimiser et d'une méthode efficace de propagation de cette borne afin de guider la recherche vers de bonnes solutions. La programmation par contraintes est en général plus efficace sur un objectif de type maximum ($\min \max_i \alpha_i x_i$) que sur un objectif de type somme ($\min \sum_i \alpha_i x_i$) parce qu'il est difficile dans le second cas de déduire d'une borne sur l'objectif des réductions de domaine pour chaque variable individuellement. En résumé, en comparaison avec les trois autres techniques, la programmation par contraintes est en général plus robuste vis-à-vis de l'ajout de contraintes additionnelles, mais moins robuste vis-à-vis des changements de taille et de caractéristiques numériques des problèmes.

La dernière classe d'algorithmes que nous étudions ici est celle des algorithmes de recherche locale [1]. Nous regroupons sous cette dénomination tous les algorithmes qui se déplacent de solution en solution voisine pour optimiser progressivement la fonction objectif. À chaque étape, l'algorithme de recherche locale considère une solution unique ou un ensemble de solutions, comme dans le cas des algorithmes génétiques [104] ou dans les approches de *scatter search* et de *path relinking* [71]. Les solutions considérées sont partielles ou complètes, irréalisables ou réalisables. Cette définition très large nous permet d'englober également les heuristiques de construction qui partent d'une solution vide et enrichissent à chaque itération une solution partielle pour obtenir en définitive une solution complète. La différence majeure entre les algorithmes de recherche locale et les trois techniques préalablement présentées concerne la complétude. Les algorithmes de recherche locale ne reposent pas sur une exploration systématique de l'espace de recherche, donc ils ne fournissent pas de preuve d'optimalité. De ce fait, les algorithmes de recherche locale sont en général rapides, efficaces et robustes vis-à-vis de la taille des problèmes car ils se préoccupent uniquement de trouver des solutions réalisables. Cependant, si les algorithmes de recherche locale produisent en général de bonnes solutions, ils n'offrent aucune garantie de trouver la solution optimale. L'efficacité d'un algorithme de recherche locale dépend de deux facteurs. Le premier facteur est la structure des voisinages définis par les opérateurs qui créent une nouvelle solution à partir d'une ou plusieurs solutions existantes. Si les solutions de bonne qualité ont en commun certaines caractéristiques qui peuvent être représentées de manière compacte et qui peuvent être conservées lorsque l'on passe d'une solution à une solution voisine, alors un algorithme de recherche locale a de bonnes chances de résoudre efficacement le problème. Ces propriétés des opérateurs

de recherche locale ne sont pas toujours stables vis-à-vis de l'ajout de contraintes additionnelles. Le deuxième facteur dont dépend l'efficacité d'un algorithme de recherche locale est sa capacité à explorer un paysage de solutions parfois accidenté de façon à atteindre des optima locaux (intensification) et à explorer des régions diverses de l'espace de recherche en ne restant pas bloqué dans un optimum local qui risque de ne pas être optimal au niveau global (diversification). L'intensification et la diversification sont en général prises en charge par les métaheuristiques comme le recuit simulé [148] ou la recherche tabou [70], mais ces fonctions peuvent être également assurées par la modification systématique des voisinages comme dans la métaheuristique des voisinages variables [76], ou par la définition et l'exploration de grands voisinages comme nous le détaillerons dans la Section 1.2.6.

1.1.3 Difficultés

Malgré l'importance des progrès attendus par le développement des algorithmes hybrides, plusieurs obstacles significatifs subsistent néanmoins pour leur déploiement plus généralisé. En effet, on peut constater que si la littérature récente compte de nombreux problèmes auxquels des algorithmes hybrides ont été appliqués avec succès, il n'en reste pas moins que ces applications ne recouvrent qu'une petite partie des cas où les algorithmes hybrides pourraient être utilisés.

La première difficulté inhérente aux algorithmes hybrides vient des coûts fixes associés à la mise en œuvre de plusieurs composantes et des coûts encourus à cause de la communication entre composantes. Il faut donc que l'hybridation apporte réellement des bénéfices pour qu'un algorithme hybride soit plus efficace qu'un algorithme pur. Ceci est particulièrement vrai pour le schéma de *recherche multiple* que nous décrirons dans la prochaine section.

La deuxième difficulté provient de la complexité (au sens ingénierie logicielle) accrue des algorithmes hybrides comparés aux algorithmes purs. D'abord, pour un algorithme hybride, il faut souvent développer et mettre en œuvre plusieurs composants logiciels (modélisations alternatives du même problème ou résolution de plusieurs sous-problèmes), ce qui nécessite plus d'efforts que le développement d'une seule technique de résolution. On pourrait d'ailleurs remarquer que la plupart des résultats expérimentaux de la littérature montrant la supériorité d'un algorithme hybride ne sont pas tout à fait objectifs dans la mesure où l'algorithme hybride a demandé en général plus de temps de développement que les algorithmes purs auxquels il est comparé. Ensuite, la mise au point d'un algorithme hybride nécessite une bonne connaissance de chacune des techniques mises en œuvre dans la coopération. Même si les remarques générales comme celles énoncées à la section précédente permettent d'avoir une idée de quelles techniques employer pour quels problèmes ou quels sous-problèmes, il est indispensable de connaître précisément les forces et les faiblesses de chacune des composantes possibles de la coopération pour créer un algorithme hybride efficace et robuste. Or, un ingénieur ou un chercheur est généralement spécialisé dans l'une de ces techniques

seulement. Enfin, les algorithmes hybrides sont en général plus difficiles à calibrer, simplement parce que le nombre de paramètres est plus grand que dans le cas des algorithmes purs.

Face à ces difficultés, il semble nécessaire de mettre au point des outils conceptuels et logiciels pour que la conception, le développement et l'application d'algorithmes hybrides soient facilités. Or, la plupart des travaux sur les algorithmes hybrides dont nous avons connaissance se concentrent sur la résolution d'un exemple particulier sans étudier la généralisation possible de la coopération mise en œuvre à d'autres problèmes, ou bien s'attachent à développer des principes généraux pour le développement d'algorithmes hybrides sans les appliquer de manière convaincante à des exemples concrets. Il serait au contraire intéressant de comprendre pourquoi un algorithme hybride s'applique particulièrement bien à un exemple précis, et d'utiliser ces informations pour étendre ce succès à des problèmes plus complexes (par exemple avec des contraintes additionnelles) ou différents, et pour savoir sur quel type de problèmes recommander l'utilisation de cet algorithme. Il serait également utile de reconnaître les points communs que partagent la multitude d'algorithmes hybrides existants, afin de mettre en évidence des schémas de coopération qui s'appliquent à de larges classes de problèmes.

1.1.4 Deux schémas génériques de coopération pour des algorithmes hybrides efficaces et robustes

Nous reprenons dans cette section la tentative d'unification et de classification des algorithmes hybrides que nous avons proposée avec Claude Le Pape [40]. À partir de nombreux exemples de la littérature, nous avons mis en évidence deux schémas génériques de coopération : la *décomposition* et la *recherche multiple*.

Le schéma de *décomposition* consiste à appliquer à un sous-problème du problème à traiter l'une des quatre techniques parmi les algorithmes spécifiques de recherche opérationnelle, le *branch-and-cut*, la programmation par contraintes et les algorithmes de recherche locale. Les informations ainsi obtenues par cette technique *esclave* sont ensuite utilisées par une autre des quatre techniques énumérées ci-dessus, le *maître*, pour résoudre le problème global. Le sous-problème donné à résoudre à l'esclave peut être un problème réduit avec moins de variables que le problème global, une approximation, une relaxation ou une version plus contrainte du problème global. Le sous-problème peut également être un problème annexe différent qui modifie l'espace sur lequel travaille le maître. Selon la définition du sous-problème, les résultats donnés par l'esclave sont utilisés par le maître pour réduire l'espace de recherche tout en garantissant de ne pas éliminer la solution optimale, ou seulement pour guider la recherche du maître vers de bonnes solutions. Les méthodes de décomposition ne sont pas spécifiques aux algorithmes hybrides. Par exemple, la décomposition de Benders, la décomposition de Dantzig-Wolfe et la relaxation lagrangienne sont des techniques de décomposition bien connues pour la programmation linéaire en nombres entiers. Nous nous intéressons ici uniquement aux cas où le sous-problème et le problème maître sont résolus par des

techniques différentes.

Dans le schéma de *recherche multiple*, au moins deux techniques d'optimisation sont appelées successivement ou en parallèle pour résoudre le problème d'origine. Dans le cas le plus élémentaire, chaque composante opère indépendamment des autres. Cependant, dans la plupart des cas, les solutions trouvées par chacune des composantes ou plus généralement l'information récoltée par chacune des composantes au cours de sa résolution est utilisée par les autres composantes pour améliorer leur propre résolution. Le schéma de recherche multiple apporte des gains en efficacité et en robustesse car l'utilisation de méthodes différentes pour résoudre le même problème introduit des capacités de diversification pour explorer des régions variées de l'espace de recherche et des capacités d'intensification pour se concentrer efficacement sur des régions prometteuses de l'espace de recherche.

Cette catégorisation des algorithmes hybrides n'est pas très détaillée, mais elle constitue une première tentative pour mettre en évidence les points communs et les différences entre les nombreux algorithmes hybrides existants. Dans la prochaine section, nous nous focaliserons sur les algorithmes hybrides où interviennent des techniques de recherche locale, dans le but de donner une vue d'ensemble plus finement structurée et relativement complète de ce sous-ensemble d'algorithmes hybrides. Avant cela, nous allons donner quelques exemples particulièrement significatifs de décomposition et de recherche multiple afin d'illustrer ces concepts. Cette courte énumération ne se veut pas exhaustive et le lecteur est invité à se référer à [40] pour un aperçu plus complet de la littérature.

Un premier exemple classique de décomposition est la génération de colonnes (pour la résolution de problèmes linéaires) ou le *branch-and-price* (pour la résolution de problèmes linéaires en nombres entiers) [12]. Cette technique est particulièrement adaptée aux modèles qui comportent un grand nombre de variables. Seul un petit nombre de ces variables prendra une valeur non nulle dans la solution optimale. Le but de la génération de colonnes est donc de ne pas expliciter toutes les variables pour réduire significativement le coût de la résolution. Autrement dit, chaque variable (colonne) représente un élément de solution et il s'agit de trouver les bonnes colonnes et de les combiner pour former une solution complète sans énumérer toutes les colonnes possibles. La génération de colonnes résout itérativement un problème maître qui consiste en un programme linéaire sur l'ensemble des colonnes déjà générées, et un sous-problème qui génère des colonnes susceptibles d'améliorer la solution courante du problème maître. En d'autres termes, le problème maître résout itérativement le problème d'origine sur un espace de recherche réduit : l'enveloppe convexe des colonnes déjà générées. Cet espace est peu à peu étendu par le sous-problème qui génère des colonnes de coût réduit négatif[1] relativement aux valeurs duales obtenues à l'itération courante du problème maître. Lorsqu'il n'existe plus de telles colonnes de coût réduit négatif, la solution optimale du

[1]Nous supposons ici et dans toute la suite de ce chapitre que nous avons à résoudre un problème de minimisation.

problème maître sur l'ensemble final de colonnes coïncide avec la solution optimale du problème sur l'espace entier de recherche. Cette technique permet de résoudre efficacement des problèmes linéaires (ou des problèmes linéaires en nombres entiers si combinée avec une recherche arborescente comme dans le *branch-and-price*) dont le nombre de variables (colonnes) est trop grand pour qu'elles soient toutes considérées explicitement dès le début de la résolution. La génération de colonnes est un schéma général de décomposition où le sous-problème peut être résolu par une technique quelconque qui n'est pas nécessairement à base de programmation linéaire. Une hybridation fréquemment employée consiste à résoudre le sous-problème avec un algorithme spécifique de recherche opérationnelle. Par exemple, pour des problèmes de routage de véhicules ou de synthèse de réseaux, le sous-problème est souvent un problème de plus court chemin contraint qui est résolu avec un algorithme de programmation dynamique [50]. Nous reviendrons plus en détail sur cet algorithme hybride dans le Chapitre 2. Une autre possibilité est de résoudre le sous-problème avec un algorithme de programmation par contraintes [54, 125]. Cette hybridation a par exemple été appliquée avec succès à des instances réelles de couverture des trajets de bus (*crew scheduling*) et d'affectation des équipages (*crew rostering*) d'une société brésilienne de transports urbains [160].

Un deuxième exemple de décomposition particulièrement efficace est rencontré dans l'utilisation de contraintes globales en programmation par contraintes [122]. Les contraintes globales permettent d'exprimer un ensemble de contraintes simples avec une modélisation de plus haut niveau et de prendre en compte simultanément cet ensemble de contraintes pour réduire ainsi plus efficacement le domaine des variables concernées. Dans la plupart des cas, la contrainte globale utilise un algorithme spécifique et polynomial de recherche opérationnelle comme algorithme de filtrage. Cet algorithme résout le sous-problème de réduction des domaines des variables de la contrainte globale, et c'est en ce sens que nous parlons ici de décomposition. Considérons l'exemple d'un ensemble de variables X pour lequel on souhaite que chacune des variables prenne une valeur différente. Cette contrainte peut être modélisée par $|X|(|X|-1)/2$ contraintes $\neq$, chacune liant deux variables et associée à un algorithme de filtrage naïf. Il est néanmoins beaucoup plus efficace de modéliser cette contrainte de manière globale par une contrainte *all-different*(X) associée à un algorithme de filtrage de couplage dans un graphe biparti [121] pour réduire simultanément les domaines de toutes les variables de X.

Un troisième exemple de décomposition est l'utilisation de relaxations en programmation par contraintes [58]. En pure programmation par contraintes, l'espace de recherche est exploré de manière arborescente avec un simple *branch-and-bound*. Les algorithmes de filtrage permettent de réduire localement l'espace de recherche (en réduisant les domaines des variables). Mais, au niveau global, la fonction objectif est le plus souvent prise en compte uniquement lorsqu'une nouvelle solution est découverte et que l'on spécifie que les solutions suivantes devront être de meilleure qualité. Or, il est difficile de propager cette contrainte sur la fonction objectif parce que le lien

avec les variables de décision est le plus souvent trop faible, sauf si la borne sur l'objectif est déjà proche de la valeur optimale ou si de nombreuses variables ont déjà été instanciées. En effet, la *backpropagation* des contraintes est souvent trop faible : il est possible d'éliminer un choix qui entraîne un coût trop élevé (pour un problème de minimisation), mais il est plus difficile d'exploiter l'information qu'un choix va coûter au moins 'x'. Ainsi, des régions sous-optimales entières sont explorées alors qu'elles pourraient être éliminées par des considérations plus sophistiquées de la fonction objectif. Utiliser une relaxation du problème permet d'élaguer certaines branches de l'arbre de recherche si l'objectif du problème relâché (une borne inférieure pour le problème initial dans le cas d'une minimisation) y est supérieur à la meilleure solution connue. La solution du problème relâché peut également guider la recherche, par exemple en suggérant des variables de branchement comme dans la méthode de *probe backtrack search* [128], ou en recommandant une manière de diviser le domaine de la variable de branchement à un nœud donné et un ordre d'exploration des nœuds fils ainsi créés [103]. La relaxation considérée est le plus souvent une relaxation linéaire, mais des relaxations plus spécifiques au problème traité peuvent également être considérées, par exemple une relaxation lagrangienne ou une relaxation semi-définie positive [146].

Un exemple de recherche multiple est l'idée du portefeuille d'algorithmes (*algorithm portfolio*) [73]. Considérons n algorithmes capables de résoudre un problème donné. Si l'un des algorithmes a des performances toujours meilleures que les $n-1$ autres, alors il est inutile de résoudre le problème avec une combinaison d'algorithmes. Cependant, il est fréquent que les performances des algorithmes varient suivant les instances à résoudre et qu'il n'existe pas d'algorithme qui soit toujours meilleur que les autres. Dans ce cas, appeler les différents algorithmes en parallèle, ou de manière séquentielle ou entrelacée (sur une machine à un seul processeur), permet d'augmenter l'efficacité et la robustesse de la résolution. Les portefeuilles d'algorithmes sont un moyen très simple d'exploiter tout type de systèmes multiprocesseurs parce que leur parallélisme ne nécessite que très peu de communications entre processeurs. Un exemple de portefeuille d'algorithmes est l'option « Concurrent » de ILOG CPLEX pour résoudre des problèmes linéaires : si l'on dispose d'un seul processeur, le simplexe dual est utilisé ; si l'on dispose de deux processeurs, un algorithme de point intérieur est aussi appelé sur le deuxième processeur ; si l'on dispose de trois processeurs, alors le simplexe primal est également appelé sur le troisième processeur. La résolution du programme linéaire est donc accélérée par ce parallélisme simple car elle est achevée dès que l'un des algorithmes a résolu le problème. L'idée de portefeuille d'algorithmes s'applique également à plusieurs versions d'un même algorithme paramétrées de manière différente, ou dans le cas d'un algorithme non-déterministe, à des initialisations différentes du générateur de nombres aléatoires.

La plupart des exemples de recherche multiple sont des algorithmes hybrides où interviennent des techniques de recherche locale, ce que nous allons détailler dans les deux sections suivantes.

1.2 Les algorithmes hybrides où interviennent des techniques de recherche locale

À l'exception de la revue récente [52], il n'existe pas à notre connaissance de tentative de taxinomie pour le sous-ensemble des algorithmes hybrides où interviennent des techniques de recherche locale. Dans cette section, nous proposons une classification différente de celle présentée par Dumitrescu et Stüzle [52]. Notre classification est articulée autour des catégories suivantes : l'accélération d'une méthode exacte, l'obtention de preuves d'optimalité, la prise en compte de contraintes additionnelles, la réduction de l'espace de recherche, la résolution exacte d'un sous-problème, la définition et l'exploration de grands voisinages. Nous avons essayé de proposer une classification aussi claire que possible, mais certains algorithmes hybrides appartiennent à plusieurs des catégories énumérées ci-dessus, donc nous avons fait le choix de ne les citer qu'une seule fois, pour leur caractéristique principale ou la plus originale.

1.2.1 Accélération d'une méthode exacte

Considérons d'abord l'un des points faibles des méthodes exactes telles que le *branch-and-cut* et comment un algorithme de recherche locale peut y remédier. Les méthodes exactes sont généralement plus lentes que les méthodes approchées et elles peuvent souvent résoudre uniquement des problèmes de taille limitée. Mais, l'exploration systématique de l'espace de recherche mise en œuvre dans les méthodes exactes peut donner de très bons résultats, en particulier si elle est guidée par une relaxation linéaire de bonne qualité (en PLNE) ou un algorithme de filtrage efficace (en programmation par contraintes). Si l'utilisateur est prêt à abandonner le fait d'atteindre et de prouver l'optimalité, alors une méthode exacte dont on a relâché quelques conditions nécessaires à l'obtention d'une preuve d'optimalité peut être un bon candidat pour un algorithme approché. La première possibilité est simplement de tronquer la recherche arborescente, par exemple d'explorer seulement un nombre donné de nœuds dans l'arbre de recherche.

Un scénario d'hybridation plus substantielle entre recherche locale et méthode exacte consiste à résoudre de manière approchée un sous-problème d'un schéma de décomposition. Par exemple, en génération de colonnes, il est nécessaire de garantir à la dernière itération du sous-problème qu'il n'existe plus de colonne de coût réduit négatif relativement aux valeurs duales courantes du problème maître si l'on souhaite obtenir une borne inférieure valide pour le problème général, et ainsi obtenir une preuve d'optimalité à la fin de la résolution. Mais, cette condition peut être levée si l'on souhaite simplement utiliser la génération de colonnes de manière heuristique. Il est même possible de conserver la complétude de la génération de colonnes en résolvant de manière approchée les premières itérations du sous-problème pour générer rapidement des colonnes de bonne qualité, puis en résolvant de manière exacte les itérations suivantes du sous-

problème pour trouver les dernières colonnes de coût réduit négatif et garantir qu'il n'en existe plus. La résolution approchée du sous-problème est en général une adaptation simple de la méthode exacte de résolution, par exemple une réduction du graphe sur lequel est calculé le plus court chemin contraint ou une réduction du nombre d'états à considérer dans le cas d'un algorithme de programmation dynamique [49]. Dans le Chapitre 2, nous montrerons comment l'adaptation d'une règle de dominance permet de résoudre de manière approchée mais plus rapidement le problème du plus court chemin contraint élémentaire. Il est également possible de résoudre le sous-problème de manière approchée avec une technique complètement différente de la résolution exacte, par exemple avec un algorithme spécifique de recherche locale. Ainsi, Rousseau *et al.* [125] utilisent différentes heuristiques de construction et plusieurs opérateurs de recherche locale de ILOG DISPATCHER pour construire l'ensemble initial de colonnes. De même, pour résoudre le problème d'élaboration de tournées de véhicules avec flotte hétérogène, Taillard [140] utilise une heuristique à mémoire adaptative pour résoudre le problème avec chaque type de véhicule considéré séparément, puis calcule la combinaison optimale de ces tournées avec un modèle de partitionnement d'ensembles. Plus généralement, Savelsbergh et Sol [130], et Xu *et al.* [157] utilisent plusieurs algorithmes heuristiques pour résoudre le sous-problème dans une approche à base de *branch-and-price* pour un problème complexe de tournées de véhicules avec une flotte hétérogène et de nombreuses contraintes additionnelles (collecte et livraison, fenêtres de temps, *etc.*).

Pour réduire de manière encore plus significative le temps de calcul nécesssaire à l'obtention d'une bonne solution, il est possible de résoudre à la fois le sous-problème et le problème maître de manière approchée. Par exemple, Gabrel *et al.* présentent un algorithme de Benders approché [63] pour la résolution d'un problème de multiflot où les coûts d'installation de capacité sur les arcs du réseau sont des fonctions en escalier avec un nombre quelconque de discontinuités. Le problème maître est un problème linéaire en variables binaires sur une formulation incomplète du polyèdre des solutions réalisables où toutes les contraintes ne sont pas explicitées. Cette formulation est enrichie itérativement en générant dans le sous-problème des contraintes qui sont violées par la solution courante du problème maître. La résolution du sous-problème est approchée dans la mesure où sont générées uniquement des contraintes appartenant à une certaine classe (inégalités de bipartition). Malgré cette première approximation, la résolution est encore coûteuse car le problème maître est un programme linéaire en variables binaires difficile. Donc, au lieu d'être résolu exactement par *branch-and-cut*, le problème maître est résolu de manière approchée avec une heuristique gloutonne (deux heuristiques sont présentées). Cet algorithme de Benders approché permet de traiter en un temps raisonnable des instances comprenant jusqu'à 50 nœuds et 90 arcs, alors que sa version exacte ne pouvait résoudre que des problèmes de moins de 20 nœuds et 40 arcs. Selon l'heuristique employée pour le problème maître, l'algorithme de Benders approché améliore de 13% à 24% en moyenne les résultats d'un algorithme de re-

cherche locale pur présenté dans le même article. De plus, sur les instances résolues par l'algorithme de Benders exact, l'erreur moyenne relative entre l'algorithme de Benders approché et la solution optimale varie de 3% et 8% selon l'heuristique employée pour résoudre le problème maître. Cet exemple montre que l'hybridation entre recherche locale et méthode exacte permet de résoudre des problèmes de plus grande taille que la méthode exacte et d'obtenir des solutions de meilleure qualité que la recherche locale.

1.2.2 Obtention de preuves d'optimalité

Considérons maintenant le défaut majeur des algorithmes de recherche locale et comment les méthodes exactes peuvent y remédier. Un algorithme de recherche locale ne donne pas de borne inférieure sur la fonction objectif à minimiser, donc ne peut pas prouver que la solution qu'il a obtenue est optimale. Pour combiner la capacité de la recherche locale à trouver des bonnes solutions rapidement et la complétude des méthodes exactes, le schéma le plus répandu est celui de la recherche multiple. L'espace de recherche est exploré systématiquement par la méthode exacte, tandis que l'algorithme de recherche locale est appelé avant ou pendant cette recherche arborescente afin de générer de bonnes solutions réalisables, de manière complètement indépendante, ou en utilisant des informations obtenues par la méthode exacte. Les heuristiques pour le *branch-and-cut* correspondent à ce type d'algorithme hybride. Certaines construisent une solution réalisable à partir de rien ; d'autres partent de la solution de la relaxation linéaire pour construire une solution respectant les contraintes d'intégralité ; d'autres encore essayent d'améliorer la solution entière courante avec des opérateurs de recherche locale. Nous décrirons ces heuristiques de manière plus détaillée dans la Section 1.3. Une hybridation similaire peut être réalisée entre des algorithmes de recherche locale et la programmation par contraintes [23].

1.2.3 Prise en compte de contraintes additionnelles

Le deuxième point faible des algorithmes de recherche locale est leur manque de robustesse vis-à-vis de l'ajout de contraintes additionnelles. Kilby, Prosser et Shaw montrent que combiner des algorithmes de recherche locale avec des techniques de programmation par contraintes permet d'atténuer cette instabilité [91]. Cette étude est menée sur des problèmes de routage de véhicules avec des contraintes de capacité, éventuellement des fenêtres de temps, et des contraintes additionnelles telles que « le client *a* et le client *b* doivent être desservis par le même véhicule », ou encore « le client *a* doit être desservi après le client *b* ». Ces contraintes additionnelles sont fréquentes dans les problèmes réels de collecte et de livraison (*pickup and delivery*). Les méthodes comparées sont d'une part des algorithmes purs de recherche locale : l'heuristique *savings* de construction de première solution, et une heuristique d'amélioration qui génère des voisins selon un certain nombre d'opérateurs classiques de recherche locale et rejette ceux qui ne sont pas valides pour toutes les contraintes. D'autre part, les algo-

rithmes hybrides étudiés sont une heuristique de construction où l'ordre d'insertion des visites est déterminé progressivement non en fonction du coût d'insertion comme dans l'heuristique *savings*, mais en fonction du nombre de positions possibles pour l'insertion de chaque visite (nombre calculé par programmation par contraintes), et une heuristique d'amélioration d'une solution existante qui consiste à enlever puis à réinsérer différemment une partie de la solution. Le choix des positions de réinsertion des visites enlevées est effectué par programmation par contraintes. Cette technique d'exploration de grands voisinages sera développée dans la Section 1.2.6. Les résultats expérimentaux de cette étude sur les jeux de données de Solomon [136] et Taillard [139] montrent que, lorsque le nombre de contraintes additionnelles augmente, les algorithmes hybrides faisant intervenir de la programmation par contraintes sont plus efficaces que les algorithmes purs de recherche locale. Ils obtiennent plus de succès dans la construction d'une première solution et améliorent plus significativement la qualité d'une solution existante.

1.2.4 Réduction de l'espace de recherche

Comme les algorithmes de recherche locale n'explorent pas l'espace de recherche de manière systématique, il est très utile de réduire leur espace de travail en excluant des régions qui ne contiennent que des solutions dont on sait par ailleurs qu'elles sont sous-optimales, ou en modélisant les voisinages de manière compacte de façon à limiter leur effort d'exploration.

Vasquez présente un exemple de coopération [151] pour résoudre le problème d'allocation de fréquences avec polarisation où la programmation par contraintes est d'abord utilisée pour réduire l'espace de recherche, puis un algorithme de recherche tabou est appelé sur cet espace réduit. Le problème étudié [47] consiste à allouer une ressource fréquentielle (fréquence et polarisation) à chaque trajet (paire origine-destination) afin de construire une solution réalisable d'une part pour les contraintes impératives d'égalité ou d'inégalité entre fréquences, écarts de fréquences et polarisations, d'autre part pour les contraintes d'écart minimal entre fréquences. La difficulté est double. Premièrement, le nombre de trajets est élevé (entre 200 et 3000) et le nombre de couples possibles (fréquence, polarisation) est très grand pour chaque trajet (entre 19 et 1000). Deuxièmement, suivant les instances, il n'existe pas toujours de solution réalisable. Ce problème de satisfiabilité est donc transformé en un problème d'optimisation où il faut satisfaire les contraintes impératives et minimiser la violation des contraintes d'écart minimal entre fréquences. La fonction de coût modélise une optimisation hiérarchique où il s'agit d'abord de trouver le plus petit niveau de relaxation k tel que toutes les contraintes de niveau k d'écart minimal entre fréquences soient satisfaites, puis de minimiser la violation des contraintes d'écart minimal entre fréquences pour les niveaux de relaxation plus stricts. Le schéma de coopération proposé est particulièrement bien adapté à ces deux difficultés. L'algorithme hybride commence d'abord par résoudre le problème pour le niveau de relaxation le plus faible,

puis s'attaque itérativement au niveau de relaxation immédiatement plus strict s'il a trouvé une solution réalisable. Pour un niveau de relaxation k fixé, la programmation par contraintes permet d'abord de réduire les domaines de chaque trajet en éliminant des valeurs (fréquence, polarisation) qui ne pourraient pas être prolongées en une solution complète réalisable. Puis l'algorithme de recherche tabou est appelé sur cet espace réduit pour effectivement construire une solution réalisable pour le niveau k. Comme les domaines de chaque trajet ont été réduits par la programmation par contraintes, chaque passage d'une solution à une solution voisine peut être effectué plus rapidement. L'algorithme de recherche locale bénéficie donc du prétraitement effectué par la programmation par contraintes car il peut explorer des configurations plus nombreuses et de meilleure qualité. À l'exception d'une seule instance, la programmation par contraintes permet de réduire les domaines de plus de 31%, et cette réduction varie entre 84 et 99% pour la moitié des instances étudiées. De plus, si pour un niveau de relaxation donné k, la programmation par contraintes réduit le domaine d'une variable au moins à l'ensemble vide et que la recherche tabou trouve une solution réalisable pour le niveau de relaxation immédiatement plus faible, alors c'est une preuve que ce niveau de relaxation contient la solution optimale. Une telle preuve est obtenue pour 25 des 30 instances étudiées. Un schéma de coopération similaire [152] a été proposé pour le problème de sac-à-dos multidimensionnel : la programmation linéaire est dans ce cas utilisée pour réduire l'espace sur lequel opère un algorithme de recherche tabou.

Une utilisation différente d'une méthode exacte pour réduire l'espace exploré par un algorithme de recherche locale consiste à décomposer le problème en séparant d'un côté les variables de décision dont les valeurs seront déterminées par l'algorithme de recherche locale, et de l'autre côté les variables secondaires qui sont certes nécessaires pour spécifier une solution complète mais dont la valeur sera déterminée par un algorithme polynomial une fois les variables de décision fixées. Cette décomposition a l'avantage de diminuer le nombre de variables considérées par l'algorithme de recherche locale, et de représenter l'espace sur lequel il évolue de manière compacte bien adaptée aux opérateurs de recherche locale. Cette hybridation est fréquemment utilisée en ordonnancement. Résoudre un problème d'ordonnancement consiste à spécifier une date de début pour chaque tâche du problème. Mais les décisions à prendre sont en fait du type « Sur la machine M, la tâche A doit-elle être exécutée avant ou après la tâche B ? ». Pour un algorithme de recherche locale, les variables de décision correspondant à ces disjonctions sont plus faciles à manipuler que les variables représentant les dates de début. En effet, les opérateurs qui intervertissent l'ordre de deux activités (ou plus) exécutées sur la même machine tels que les opérateurs de réparation [23] ou de *shifting bottleneck* [4] s'appliquent particulièrement bien à cette modélisation. Une fois les variables disjonctives fixées, un algorithme polynomial détermine les dates de début pour chaque tâche de façon à ce que l'ordonnancement ainsi spécifié ait un coût optimal (optimal étant donné les décisions fixées). Cet algorithme polynomial est par exemple un algorithme de plus long chemin s'il s'agit de minimiser la date de fin de l'ordonnan-

cement [147], un programme linéaire s'il s'agit de minimiser des coûts d'avance et de retard [14], ou un algorithme de programmation dynamique lorsque la fonction objectif est une somme de fonctions linéaires par morceaux plus générales [78, 79].

Le dernier algorithme hybride que nous mentionnerons dans ce paragraphe se distingue des deux précédents dans la mesure où c'est maintenant l'algorithme de recherche locale qui est utilisé pour réduire l'espace de recherche de la méthode exacte[2]. Focacci et Shaw présentent en effet les algorithmes de recherche locale comme une extension des règles de dominance utilisées pour élaguer l'arbre de recherche en programmation par contraintes [59]. Pour un problème de minimisation, les règles de dominance stipulent que si une solution partielle s peut être étendue pour former une solution complète dont le coût est inférieur à l'extension optimale d'une solution partielle s', alors toutes les extensions de s' peuvent être éliminées, ce qui permet de réduire significativement l'espace de recherche. Cependant, éliminer s' peut être contre-productif si le sous-arbre dont s est la racine n'a pas encore été exploré. En effet, en explorant le sous-arbre dont s' est la racine, on pourrait obtenir rapidement une extension de s' qui améliore la meilleure solution connue et ainsi restreindre immédiatement la recherche à suivre. L'application systématique des règles de dominance peut ainsi ralentir la découverte de bonnes solutions. Focacci et Shaw proposent donc d'éliminer uniquement les solutions partielles qui ne peuvent pas être prolongées en des solutions complètes de coût inférieur à la meilleure solution connue. La solution s' est donc traitée comme un conflit (*no-good*), c'est-à-dire une solution partielle qui ne peut pas être prolongée en une solution complète. En effet, toute prolongation de s' violerait la borne supérieure sur le coût imposée par la meilleure solution connue. N'importe quel algorithme peut être utilisé pour montrer qu'une solution partielle est dominée, notamment un algorithme de recherche locale. Pour l'exemple du problème de voyageur de commerce [59], c'est une heuristique de plus proche voisin et un opérateur de recherche locale spécifique qui sont employés. Si cet algorithme découvre que le problème du prolongement de la solution partielle considérée p est une relaxation du problème de prolongement défini par l'un des conflits c en mémoire, alors le sous-arbre dont p est la racine peut être élagué. Plus spécifiquement, si l'algorithme de recherche locale prolonge c de façon à obtenir un chemin hamiltonien qui contient les mêmes nœuds que p et dont les nœuds de départ et d'arrivée sont les mêmes que p, mais de telle façon que le coût correspondant à ce prolongement de c est inférieur à celui de p, alors p est éliminé. Comparé à une approche pure de programmation par contraintes et à l'application systématique des règles de dominance, cet algorithme hybride diminue très significativement le temps de résolution et le nombre d'échecs de la recherche arborescente sur de nombreuses instances de voyageur de commerce symétriques et asymétriques.

[2] Cet algorithme pourrait aussi être classé dans le groupe des algorithmes d'accélération d'une méthode exacte présenté à la Section 1.2.1.

1.2.5 Résolution exacte d'un sous-problème

Les techniques de recherche locale comprennent en général plusieurs opérations successives telles que la sélection, la mutation et le croisement des individus pour les algorithmes génétiques, ou plus généralement encore s'articulent autour de stratégies de diversification et d'intensification. Les algorithmes de recherche locale pure résolvent souvent ces sous-problèmes de manière approchée, et même dans un certain nombre de cas comme des problèmes de satisfaction de contraintes et non d'optimisation. Or, chacun de ces sous-problèmes pourrait être résolu avec un algorithme exact pour obtenir des heuristiques probablement plus coûteuses, mais aussi plus efficaces.

Par exemple, la plupart des stratégies de diversification visent à trouver une solution « assez différente » du dernier optimum local, mais formalisent rarement ce problème sous la forme d'un problème d'optimisation où il s'agirait par exemple de minimiser la dégradation de l'objectif tout en garantissant un certain éloignement de l'optimum local. Dans le Chapitre 2, nous proposerons une stratégie de diversification qui utilise un algorithme exact pour trouver (en cas de succès) une solution toujours meilleure que le dernier optimum local.

Un deuxième exemple est celui du croisement des individus dans le cadre des algorithmes génétiques. Au lieu de se contenter de trouver un seul enfant qui partage les caractéristiques de ses deux parents, un algorithme exact peut être utilisé pour déterminer le meilleur individu parmi tous les individus qui partagent les caractéristiques des deux parents. Yagiura et Ibaraki utilisent notamment un algorithme de programmation dynamique pour résoudre ce sous-problème de croisement [159]. Cette hybridation est très proche de l'idée d'exploration de grands voisinages que nous détaillerons dans la section suivante.

Un troisième exemple particulièrement intéressant est celui du choix du prochain voisin exploré. En recherche locale pure, le voisin vers lequel se déplace l'algorithme d'une itération à l'autre est choisi de manière très simple : les solutions voisines de la solution courante sont énumérées jusqu'à ce que l'on trouve une solution qui remplisse les critères souhaités (meilleur coût, pas de composante tabou, ...), ou jusqu'à ce qu'un critère d'arrêt soit rencontré (nombre maximum de voisins énumérés par exemple). Cette énumération correspond en fait à la stratégie connue sous le nom de *generate and test* en programmation par contraintes. Or, tous les travaux sur l'arc-consistance, notamment [106], ont montré que cette stratégie était facilement supplantée par des algorithmes plus sophistiqués de filtrage et de propagation. Il est donc naturel d'utiliser ces mêmes algorithmes pour explorer un voisinage de recherche locale afin d'améliorer les performances obtenues par simple énumération. Pesant et Gendreau [115] décrivent un tel algorithme de coopération entre recherche locale et programmation par contraintes pour résoudre le problème du voyageur de commerce avec fenêtres de temps. Le voisinage utilisé est de type 3-OPT : si la solution courante emprunte les arcs (I, I^+), (J, J^+) et (K, K^+) alors l'opérateur de recherche locale la transforme en une solution où ces 3 arcs sont remplacés par les arcs (I, J^+), (K, I^+) et (J, K^+). L'avantage

de cet opérateur est qu'il préserve l'orientation de tous les autres arcs de la solution, ce qui est particulièrement utile en présence de fenêtres de temps. Cependant, pour un problème à n villes, cet opérateur définit $O(n^3)$ voisins pour chaque solution, donc il n'est pas réaliste d'explorer le voisinage ainsi défini par énumération exhaustive comme c'est souvent le cas pour des voisinages plus petits. De plus, parmi ce grand nombre de voisins, tous ne sont pas réalisables (pour les contraintes temporelles) et seul un petit nombre ont un coût meilleur que la solution courante. Donc, Pesant et Gendreau proposent d'explorer ce voisinage à l'aide de la programmation par contraintes. Les variables du sous-problème sont les nœuds I, J et K, dont les domaines sont initialement $\{1,...,n\}$ et sont progressivement réduits jusqu'à l'obtention du triplet qui définit le voisin réalisable de meilleur coût. La réduction des domaines est accélérée par le calcul d'une borne inférieure sur le coût d'un échange 3-OPT partiel (lorsque seulement I, ou bien seulement I et J sont instanciés). Sur des exemples de problèmes nombreux et variés, les résultats numériques montrent qu'en moyenne 3% et au plus 18% seulement des voisins sont réalisables, ce qui justifie de ne pas explorer le voisinage par énumération exhaustive. Pour l'algorithme de programmation par contraintes avec borne inférieure, le ratio du nombre de retours en arrière (*backtracks*) divisé par le nombre de voisins est en moyenne de 9% et au plus de 29%. De plus, ce ratio est parfois même inférieur à la proportion de solutions réalisables dans le voisinage, ce qui montre que le calcul de bornes inférieures est efficace pour éliminer des régions sous-optimales de l'espace de recherche. L'utilisation d'un algorithme sophistiqué pour explorer un voisinage de taille polynomiale certes, mais plus grand que la plupart des voisinages utilisés habituellement en recherche locale pure, fait de l'algorithme de Pesant et Gendreau un précurseur des algorithmes à base de grands voisinages.

1.2.6 Définition et exploration de grands voisinages

Comme nous l'avons mentionné à la Section 1.1.2, la réussite d'un algorithme de recherche locale dépend fortement des caractéristiques de ses voisinages. Si un voisinage est petit, autrement dit, si les opérateurs de recherche locale ne définissent qu'un petit nombre de voisins pour chaque solution, alors le nombre d'optima locaux sera élevé et un grand nombre de déplacements sera nécessaire pour passer d'un optimum local à un autre. Il sera donc à la fois difficile de s'échapper d'un optimum local, et difficile d'explorer des parties très différentes de l'espace de recherche. Au contraire, si le voisinage est grand, alors la difficulté principale résidera dans l'exploration efficace de ce voisinage. Dans ce paragraphe, nous décrirons un certain nombre d'algorithmes hybrides récents qui exploitent à des degrés divers les idées de définition et d'exploration de grands voisinages. Nous donnerons également une caractérisation qui unifie cette classe d'algorithmes. Nous mettrons enfin en évidence une question essentielle pour la généralisation de tels algorithmes à laquelle nous apporterons une réponse dans le Chapitre 3.

Pour illustrer l'idée de grands voisinages, commençons par un exemple, l'algo-

rithme de Shaw pour le problème de routage de véhicules avec fenêtres de temps [135] qui a introduit le terme de *large neighborhood search* (LNS). Cette approche consiste à résoudre une série de sous-problèmes où il s'agit à chaque fois de réinsérer différemment des clients qui ont été retirés de la solution courante. En d'autres termes, il faut résoudre le problème général étant donné qu'un certain nombre de décisions ont été fixées (par quel véhicule et dans quel ordre sont visités certains clients). La meilleure solution trouvée pendant la résolution du sous-problème devient la nouvelle solution courante, à partir de laquelle est défini un nouveau sous-problème et ainsi de suite. Le sous-problème est résolu par programmation par contraintes. Cependant, le voisinage défini par le sous-problème étant de taille potentiellement exponentielle (exponentielle en k, où k est le nombre de décisions non-fixées à chaque étape, qui peut varier), il est exploré de manière heuristique en tronquant la recherche arborescente avec une limite sur le nombre de branches droites explorées (*limited discrepancy search [77]*).

La technique de *large neighborhood search* que nous avons illustrée sur cet exemple repose sur deux idées fondamentales, l'une pour la définition, l'autre pour l'exploration de grands voisinages, comme nous l'avons mis en évidence dans [42]. Le premier point commun des approches à base de LNS est que les voisinages sont définis par le fait de fixer une partie de la solution courante. Les éléments de la solution qui sont fixés sont des variables implicites ou explicites du modèle. Par exemple, pour un problème d'ordonnancement, on peut fixer les dates de début de certaines activités (variables explicites) ou ajouter des contraintes additionnelles pour imposer à une activité d'être ordonnancée avant une autre activité (variables disjonctives implicites). Les autres variables sont relâchées : elles sont libres de changer de valeur. Le voisinage est ainsi défini par toutes les extensions de la solution partielle fixée. Comme un nombre significatif de variables sont relâchées simultanément, les voisinages utilisés en LNS sont en général de grande taille, typiquement plus grands que les voisinages utilisés habituellement en recherche locale, et souvent de taille exponentielle. Le deuxième point commun des approches à base de LNS est que les voisinages ainsi définis nécessitent un algorithme puissant pour être explorés efficacement ; en particulier, il n'est pas possible de les explorer par énumération ou avec des heuristiques naïves. Une possibilité couramment employée est d'explorer ces voisinages avec une forme de recherche arborescente comme la programmation par contraintes ou le *branch-and-cut*. Cette recherche est le plus souvent tronquée par une limite en temps, en nombres de nœuds ou en nombre de branches droites explorées. Ces deux idées de définition et d'exploration de grands voisinages sous-tendent sous un autre nom un certain nombre d'approches similaires telles que *referent domain optimization* [70], *russian doll search* [154], une hybridation entre recherche tabou et *branch-and-bound* [19], Mimausa [102], Popmusic [123], *large scale neighborhood search* [5], *iterated local search* [101] et dans une moindre mesure la métaheuristique des voisinages variables [76].

Ces techniques de définition et d'exploration de grands voisinages ont été appliquées avec succès à des problèmes aussi divers que le routage de véhicules [158,

135, 16, 126, 117], l'allocation de fréquences [111], la synthèse et le routage dans les réseaux de communication [28, 114], la localisation [141] et l'ordonnancement. Par exemple, pour minimiser le temps total d'exécution pour le problème d'ordonnancement d'atelier, l'opérateur de *shifting bottleneck* [4] consiste à fixer itérativement les séquences d'activités sur toutes les machines sauf une (le goulet d'étranglement), puis à ordonner les activités sur cette machine grâce à un algorithme spécifique d'ordonnancement pour le problème à une machine [20]. L'opérateur *shuffle* [7] généralise l'opérateur de *shifting bottleneck* en relâchant les décisions d'ordonnancement sur plusieurs machines simultanément et en résolvant le sous-problème résultant par *edge-finding* [21]. D'autres approches plus récentes où les voisinages sont explorés par programmation par contraintes donnent également de bons résultats sur des problèmes d'ordonnancement plus généraux [23, 110, 24, 22, 112]. La diversité du champ applicatif des grands voisinages se retrouve dans les méthodes employées pour résoudre les sous-problèmes. Ils sont très souvent résolus par programmation par contraintes, mais aussi par programmation linéaire [158], *branch-and-cut* [111], avec une série d'heuristiques [117] ou avec des algorithmes spécifiques de recherche opérationnelle [4, 7, 5].

Notons enfin l'algorithme de fusion de tours (*tour merging*) [36] (voir aussi la thèse de Schilham [132] pour une approche similaire) qui utilise indirectement des grands voisinages pour résoudre le problème du voyageur de commerce. Si l'on dispose d'une heuristique non-déterministe pour résoudre un problème, une manière simple d'améliorer ses performances est de l'appliquer plusieurs fois avec des initialisations différentes du générateur de nombres aléatoires et de sélectionner la meilleure solution parmi toutes les solutions obtenues. Cependant, on perd ainsi de nombreuses informations, par exemple que certaines composantes apparaissent plus souvent que d'autres dans des solutions de bonne qualité. L'algorithme de fusion de tours [36] utilise cette information de la manière suivante. Une heuristique est dans un premier temps appelée plusieurs fois sur le problème global, et dans un deuxième temps sur le graphe de support des solutions ainsi obtenues : si un arc reliant deux villes n'est présent dans aucune des solutions obtenues dans la première phase, alors il est exclu du graphe étudié dans la deuxième phase. Le temps nécessaire à la fusion des tours dans la deuxième phase est typiquement une fraction du temps passé dans la première phase et la solution obtenue dans la deuxième phase est significativement meilleure que la meilleure des solutions obtenues au cours de la première phase. Cette méthode en deux phases s'apparente aux techniques de réduction de l'espace de recherche présentées dans la Section 1.2.4. Mais c'est aussi un algorithme de grands voisinages. En effet, la deuxième phase peut être interprétée comme la résolution d'un sous-problème où l'on a fixé négativement de nombreuses décisions « Cet arc est-il emprunté dans la solution ? ». Le sous-problème ainsi défini a la même structure que le problème général et est encore de grande taille.

Les algorithmes de grands voisinages présentés dans la littérature sont en général efficaces et robustes. Leur point faible est qu'ils ne sont pas génériques. En effet, ils exploitent la plupart du temps la structure de haut niveau du problème étudié pour

définir les voisinages explorés, c'est-à-dire pour choisir les variables à fixer à chaque itération. Comme la grande taille des voisinages et la puissance des algorithmes utilisés pour résoudre les sous-problèmes apportent aux approches à base de LNS des propriétés intrinsèques de diversification et d'intensification, la définition des voisinages est une question fondamentale pour ces approches, encore plus que pour les algorithmes habituels de recherche locale. Or, pour définir un voisinage intéressant, il faut que les variables relâchées simultanément soient liées. En effet, le problème peut être tellement contraint que, même si un nombre important de variables est relâché, il n'existe pas d'autre extension de la solution partielle définie par les fixations que la solution courante. Il est donc essentiel de relâcher ensemble des variables proches pour que la liberté introduite par le relâchement de l'une permette à une autre de changer de valeur et réciproquement. Ensuite, en comparaison avec des voisinages de plus petite taille, l'essence du LNS est d'opérer des changements plus complexes qui produisent des solutions plus éloignées de la solution courante. C'est ce qui apporte de la diversification et ce qui permet de résoudre le noyau dur des problèmes. Mais, pour qu'un algorithme de grands voisinages soit particulièrement efficace (plus efficace qu'un algorithme de recherche locale classique), il faut que le voisinage qu'il définit corresponde à un noyau indivisible du problème initial et ne puisse donc pas être décomposé en une série de problèmes indépendants plus petits. En effet, le LNS repose sur l'espoir que le gain obtenu en calculant simultanément de nouvelles valeurs pour les variables relâchées sera plus grand que le gain obtenu en changeant indépendamment la valeur de chacune de ces variables. Il est donc essentiel de relâcher ensemble des variables proches pour qu'elles puissent prendre des nouvelles valeurs qui sont globalement meilleures que celles de la solution courante. Les algorithmes existants utilisent trois grandes stratégies pour définir leurs voisinages. La stratégie la plus fréquente est d'exploiter la structure de haut niveau du problème à résoudre pour déterminer quelles variables sont proches : les voisinages sont spécifiques au problème étudié. Par exemple, pour déterminer quels clients enlever et réinsérer à chaque itération, Shaw [135] utilise une fonction de proximité entre chaque paire de clients, calculée à partir de la distance géographique entre les clients et le fait qu'ils sont ou non desservis par le même véhicule dans la solution courante. La deuxième stratégie consiste à choisir aléatoirement les décisions à fixer à chaque itération. Utilisée telle quelle, cette stratégie donne parfois des résultats acceptables si l'exploration de chaque voisinage est strictement limitée [114] car elle s'apparente alors aux stratégies de redémarrage rapide [74]. Mais elle est le plus souvent utilisée en combinaison avec une stratégie qui utilise la structure du problème : des voisinages semi-aléatoires permettent d'introduire simplement de la diversification. Enfin, l'algorithme de fusion de tours [36] introduit une nouvelle stratégie de définition des voisinages qui repose sur l'utilisation non seulement de la solution courante, mais aussi d'un certain nombre de solutions découvertes auparavant dans l'exploration. Actuellement, il n'existe donc pas de stratégie de définition de grands voisinages qui soit applicable sans modification à tout type de problème. La stratégie de Cook et

Seymour [36] nous semble la plus prometteuse pour créer un algorithme générique de grands voisinages et c'est ce que nous montrerons dans le Chapitre 3.

1.3 Les heuristiques pour le branch-and-cut

1.3.1 Intérêt

Lorsqu'un modèle de programmation linéaire en nombres entiers est résolu par *branch-and-cut*, les solutions entières sont théoriquement obtenues par simple exploration arborescente, par exemple avec le schéma de branchement suivant qui est le plus couramment utilisé. À chaque nœud, s'il existe au moins une variable x qui prend une valeur fractionnaire f dans la relaxation continue, alors on branche sur cette variable, en imposant sur une branche $x \leq \lfloor f \rfloor$, et sur l'autre branche $x \geq \lceil f \rceil$, ce qui revient dans le cas particulier d'une variable binaire à fixer x respectivement à 0 et 1. L'espace de recherche est ainsi partitionné et la solution de la relaxation au nœud courant est exclue. Le détail de l'algorithme de *branch-and-cut* est donné en page 34. Lorsque la profondeur dans l'arbre augmente, on s'attend donc à ce que le nombre de variables fractionnaires diminue et on espère que la relaxation deviendra rapidement entière. Cependant, en pratique, il est parfois difficile de trouver les nœuds où la relaxation est entière.

Prenons comme premier exemple celui des contraintes de type *generalized upper bound* (GUB) : $\sum_{i \in I} x_i = 1$, où chaque $x_{i, i \in I}$ est une variable binaire. Si x_{i_0} est la variable de branchement, la contrainte supplémentaire $x_{i_0} = 0$ qui sera ajoutée à l'un des nœuds fils ne changera en général pas fondamentalement la solution de la relaxation par rapport au nœud courant. Dans l'autre branche, la contrainte $x_{i_0} = 1$ permettra au contraire de fixer toutes les autres variables de la contrainte à zéro. L'arbre sera ainsi déséquilibré et le choix d'une bonne variable de branchement est alors équivalent au choix de la variable x_i à fixer à 1, donc aussi difficile. Pour éviter de brancher de nombreuses fois sans progresser, il est posssible de brancher par dichotomie [99] : si I' est un sous-ensemble de I tel que $0 < \sum_{i \in I'} x_i < 1$ dans la relaxation, alors on impose sur l'une des branches $\sum_{i \in I'} x_i = 0$, et sur l'autre $\sum_{i \in I'} x_i = 1$. Ainsi, l'arbre est moins déséquilibré, du moins si le sous-ensemble I' est choisi de telle sorte que chacune des deux contraintes ajoutées lors du branchement soit compatible avec les autres contraintes du problème, ce qui n'est pas trivial. Un deuxième exemple où la recherche arborescente rencontre des difficultés est celui des symétries. Si le problème comporte un grand nombre de symétries, alors il est possible que les deux nœuds fils issus d'un même nœud père soient symétriques l'un de l'autre, et que chaque nœud définisse un problème comportant lui-même encore un grand nombre de symétries. Dans ce cas, l'approche « diviser pour régner » de l'exploration arborescente ne permet pas de résoudre plus facilement le problème. Plus généralement, si la relaxation continue est une approximation médiocre de l'enveloppe convexe des solutions entières du problème, alors il sera difficile

Considérons : un problème de minimisation.
Notons : S l'ensemble de nœuds ouverts.
Notons : UB (respectivement LB) la meilleure borne supérieure (resp. inférieure) connue à un instant t sur l'objectif à minimiser.
Notons : $LB(n)$ la borne inférieure locale calculée au nœud n.

1: S $\leftarrow$ { nœud racine }
2: $UB \leftarrow +\infty$
3: $LB \leftarrow -\infty$
4: **tant que** $S \neq \emptyset$ **faire**
5: Choisir un nœud n dans S.
6: {*début de la phase de coupe*}
7: **tant que** certains critères d'arrêt sur la génération de coupes ne sont pas remplis **faire**
8: Résoudre la relaxation continue au nœud n.
9: **si** cette relaxation n'est pas réalisable **alors**
10: Aller à l'étape 33. {*destruction du nœud*}
11: **sinon**
12: Notons s le vecteur solution de la relaxation et v sa valeur.
13: $LB(n) \leftarrow v$
14: **si** $v \geq UB$ **alors**
15: Aller à l'étape 33. {*destruction du nœud*}
16: **sinon si** toutes les variables entières prennent une valeur entière dans s **alors**
17: $UB \leftarrow v$
18: $S \leftarrow S \setminus \{$nœuds $n \in S$ tels que $LB(n) \geq UB\}$
19: Aller à l'étape 33. {*destruction du nœud*}
20: **sinon**
21: Ajouter des coupes pour renforcer la formulation.
22: **fin si**
23: **fin si**
24: **fin tant que**
25: {*fin de la phase de coupe*}
26: {*début de la phase de branchement*}
27: Choisir une variable entière x qui prend une valeur fractionnaire f dans s.
28: Créer le nœud n_1 avec la contrainte additionnelle $x \leq \lfloor f \rfloor$ et le nœud n_2 avec la contrainte additionnelle $x \geq \lceil f \rceil$
29: $S \leftarrow S \cup \{n_1, n_2\}$
30: $LB(n_1) \leftarrow v$
31: $LB(n_2) \leftarrow v$
32: {*fin de la phase de branchement*}
33: $S \leftarrow S \setminus \{n\}$
34: $LB \leftarrow \min_{n \in S} LB(n)$
35: **fin tant que**

Algorithme 1: Squelette de l'algorithme de *branch-and-cut*.

d'utiliser la relaxation continue pour guider la recherche dans l'arbre et trouver des nœuds où la relaxation continue est entière.

De plus, la résolution d'un programme linéaire en nombres entiers par *branch-and-cut* est toujours une compétition entre des stratégies qui visent à obtenir rapidement des solutions entières de bonne qualité et des stratégies qui visent à obtenir la solution optimale et prouver son optimalité en un temps le plus court possible. Cette compétition se traduit en particulier dans le choix des variables de branchement. Pour obtenir une solution entière rapidement, il est intéressant de prendre des décisions « évidentes » : par exemple lorsqu'une variable binaire vaut 0.999 dans la relaxation, suivre la branche qui la fixe à 1. Au contraire, pour prouver l'optimalité, il est souvent préférable de choisir en haut de l'arbre des variables de branchement dont la fixation a un fort impact, sinon la taille de l'arbre risque d'exploser. Or, il est intéressant d'obtenir une solution entière de qualité en soi, mais la valeur de cette solution permet aussi d'accélérer la preuve d'optimalité, par exemple en permettant d'élaguer les branches de l'arbre où la relaxation est supérieure à cette valeur (étapes 14, 15 et 18 de l'algorithme de *branch-and-cut*), ou de fixer certaines variables par des considérations sur leur coût réduit (*reduced cost fixing* [108]). C'est pourquoi il est utile de combiner l'exploration arborescente avec d'autres algorithmes, par exemple des techniques de recherche locale, pour trouver des solutions entières si possible assez tôt dans la recherche et sans polluer l'arbre de recherche global. Cette dissociation permet également de mettre en œuvre plusieurs heuristiques différentes à chaque nœud. Les heuristiques sont aujourd'hui une composante essentielle des solveurs commerciaux de *branch-and-cut*. Par exemple, on constate expérimentalement que la moitié environ des solutions entières trouvées par ILOG CPLEX proviennent des heuristiques, tandis que la moitié sont des solutions spontanément entières de relaxations continues.

La plupart des heuristiques décrites dans la littérature sont spécifiques à un problème particulier, par exemple un problème d'ordonnancement où la solution de la relaxation est utilisée pour ordonner les activités [25]. Ici, nous décrirons uniquement les heuristiques qui peuvent s'appliquer à tout problème linéaire en nombres entiers, quelle que soit sa structure. Pour trouver une solution entière et réalisable, deux grandes stratégies sont possibles. La première est de partir d'une solution réalisable mais fractionnaire, typiquement la solution de la relaxation continue, et de procéder par arrondis successifs pour atteindre l'intégralité. La deuxième est de partir d'une solution entière mais irréalisable et d'énumérer les solutions entières voisines afin d'atteindre la réalisabilité vis-à-vis des contraintes linéaires du problème. Nous décrirons dans les deux prochaines sections les principaux algorithmes qui correspondent à ces deux stratégies. Nous présenterons ensuite les heuristiques qui procèdent par énumération des points extrêmes d'un polyèdre donné. Les heuristiques de ces trois classes n'utilisent de fait que l'aspect linéaire de la programmation linéaire en nombres entiers (pour les classes 1 et 3) ou que son aspect combinatoire (pour la classe 2). Nous introduirons à la fin de cette partie deux heuristiques plus récentes qui exploitent simultanément ces deux as-

pects en utilisant la recherche arborescente du *branch-and-cut* par la mise en œuvre cette technique sur un sous-problème ou l'utilisation des informations récoltées à différents nœuds de l'arbre de *branch-and-cut*. Ces heuristiques représentent un premier pas vers l'intégration de techniques de recherche locale au *branch-and-cut*, et vers l'application à la programmation linéaire en nombres entiers des idées de la communauté des algorithmes hybrides que nous avons synthétisées dans les Sections 1.1 et 1.2.

1.3.2 Heuristiques d'arrondis

Une idéee naturelle pour produire une solution entière réalisable est de partir de la solution optimale de la relaxation continue, puisque son obtention est la première étape de l'algorithme de *branch-and-cut*, et de l'arrondir afin d'atteindre une solution entière tout en conservant la réalisabilité pour toutes les contraintes linéaires. Cependant, il s'avère que, pour des modèles sans structure particulière, un arrondi brutal tel que l'arrondi de chaque variable fractionnaire à l'entier le plus proche ne produit que très rarement une solution réalisable. Il est donc indispensable de guider le processus d'arrondi, ce qui peut être accompli de plusieurs manières.

Heuristiques de plongeon

Une première approche est celle des heuristiques de plongeon [18], plus connues sous le nom de *diving heuristics* ou *hard variable fixing heuristics* [57]. Ces algorithmes partent de la solution optimale de la relaxation continue, arrondissent certaines variables à des valeurs entières, en déduisent des fixations supplémentaires pour certaines des variables entières libres restantes (par des techniques de *presolve* [81, 129] ou de fixation par les coûts réduits [108]), puis résolvent éventuellement à nouveau le programme linéaire. Ces différentes phases sont itérées jusqu'à obtention d'une solution entière réalisable (quand toutes les variables sont fixées), ou jusqu'à découverte d'une irréalisabilité ou d'une fonction objectif supérieure à la meilleure solution connue (pour un problème de minimisation). Plusieurs variations à partir de ce schéma général permettent d'obtenir de nombreuses heuristiques différentes. La première variation est la fréquence de résolution du programme linéaire, ce qui permet d'obtenir des heuristiques rapides, ou des heuristiques plus coûteuses mais avec un taux de succès plus élevé. La deuxième variation est l'ordre dans lequel les variables sont fixées. Cet ordre dépend par exemple du statut des variables (en base ou hors base), de leur valeur dans la relaxation (on peut par exemple fixer d'abord les variables dont les valeurs sont proches de 0 ou de 1) ou de leur apparition dans des contraintes particulières. La troisième variation concerne la valeur à laquelle les variables sont arrondies. En effet, arrondir à l'entier le plus proche est parfois inefficace. Par exemple, en présence d'une contrainte de type $\sum_{i \in I} x_i = 1$, il est possible que toutes les variables x_i prennent dans la relaxation une valeur proche de $1/|I|$, donc inférieure à 0.5 si $|I| > 2$. Dans ce cas, quel que soit l'ordre de fixation des variables, l'arrondi à l'entier le plus proche fixera

une ou plusieurs des variables x_i à zéro, ce qui ne modifiera pas significativement la solution de la relaxation continue, donc ne permettra pas de guider plus efficacement les arrondis suivants, et risquera même en définitive de ne pas produire une solution réalisable. Dans ce cas, il faut arrondir l'une des variables x_i à 1 même si sa valeur dans la relaxation est plus proche de 0. De nombreuses heuristiques de plongeon sont implémentées dans le logiciel ILOG CPLEX [18].

Arrondi aléatoire

Une deuxième approche est celle de l'arrondi aléatoire ou *randomized rounding* [120]. Cette stratégie consiste à arrondir la solution de la relaxation linéaire de la manière suivante. Considérons le programme linéaire en variables binaires[3] suivant :

$$(IP) \quad \begin{cases} \min \sum_{j=1}^{n} c_j x_j & \\ \sum_{j=1}^{n} a_{ij} x_j \geq b_i & \forall\, i \in \{1, \ldots, m\} \\ x_j \in \{0,1\} & \forall\, j \in \{1, \ldots, n\} \end{cases}$$

et notons (LP) sa relaxation continue. Si x_j^* est la valeur de la variable x_j dans la solution optimale de (LP), alors x_j est arrondie à 1 avec la probabilité x_j^* et arrondie à 0 avec la probabilité $1 - x_j^*$. Notons $\hat{x}$ la solution ainsi obtenue. L'espérance de $c\hat{x}$ est égale à la valeur de la solution optimale du programme linéaire cx^*. De plus, pour chaque contrainte, l'espérance de $A\hat{x}$ est supérieure à b, donc, en espérance, $\hat{x}$ est réalisable. Naturellement, c'est uniquement sur des modèles de structure très particulière que l'on peut obtenir des résultats d'existence avec une probabilité non nulle (ou avec une probabilité de $1 - \varepsilon$) d'une solution effectivement réalisable dont le coût est borné par une fonction de la valeur de l'optimum continu cx^* (et aussi du coefficient ε dans le deuxième cas). Par exemple, pour les problèmes où la matrice A et le vecteur de coût c n'ont que des coefficients de valeur 0 ou 1, une variante de cet algorithme fait intervenir un coefficient multiplicatif de changement d'échelle : x_j est arrondie à 1 avec la probabilité δx_j^* et arrondie à 0 avec la probabilité $1 - \delta x_j^*$ pour un certain coefficient $\delta > 1$. L'idée est ici d'augmenter la probabilité que les contraintes $Ax \geq b$ soient vérifiées. Raghavan et Thompson [120] utilisent notamment cette variante pour donner un résultat d'existence avec probabilité non nulle pour ce type de modèle. Cependant, si les garanties de performance exhibées pour les algorithmes d'arrondi aléatoire sont toujours intéressantes en théorie, en pratique elles ne sont intéressantes que si la borne sur le coût de la solution arrondie est proche du coût de la solution optimale de la relaxation, ce qui n'est obtenu que très rarement.

Utilisation de deux solutions fractionnaires

Une troisième approche consiste à ne pas considérer une seule solution fractionnaire, mais à utiliser la solution optimale de la relaxation et une solution à l'intérieur du

[3] La plupart des heuristiques que nous décrivons peuvent être étendues au cas de variables entières générales.

polyèdre des solutions réalisables. Considérons toujours le même programme linéaire en nombres entiers (IP) et notons B l'ensemble des variables x en base dans la solution optimale x^* de (LP). Si l'on remplace le membre de droite b_i par $b'_i = b_i + \frac{1}{2}\sum_{j\in B}|a_{ij}|$ pour toutes les contraintes qui sont saturées en x^*, alors on peut montrer facilement que le vecteur obtenu en arrondissant à l'entier le plus proche chaque variable de la solution optimale x' de ce nouveau programme linéaire sera réalisable pour toutes les contraintes saturées en x^*. Une des heuristiques proposées par Hillier [80] parcourt donc le segment $[x^*, x']$ en arrondissant à l'entier le plus proche les vecteurs $(1-\alpha)x^* + \alpha x'$ pour les différentes valeurs de $0 \leq \alpha \leq 1$ qui déclenchent un changement dans la solution arrondie. Faaland et Hillier [55], et Ibaraki *et al.* [85] proposent des heuristiques similaires. D'autres stratégies [131, 105, 118] utilisent un algorithme de point intérieur pour résoudre le programme linéaire et obtenir des points à l'intérieur du polyèdre qui servent ensuite de solution initiale pour des algorithmes de recherche locale. Une stratégie d'arrondi assez similaire est celle de l'*arrondi directionnel* [66]. Considérons à nouveau deux vecteurs x^* et x', appelés dans la terminologie de l'arrondi directionnel le point focal et le point de base. Comme dans les heuristiques mentionnées précédemment dans ce paragraphe, typiquement ce sont respectivement la solution optimale de (LP) et un point à l'intérieur du polyèdre. Alors, l'arrondi directionnel $\delta(x^*, x')$ est défini composante par composante par :

$$\delta(x^*, x')_j = \begin{cases} 1 & \text{si } x'_j > x^*_j \\ 0 & \text{si } x'_j < x^*_j \\ x^*_j & \text{si } x'_j = x^*_j \text{ et } x^*_j \in \{0,1\} \\ \lfloor x^*_j + \frac{1}{2} \rfloor & \text{sinon} \end{cases}$$

Glover *et al.* [72] utilisent cette stratégie avec une succession de différents vecteurs x' pour obtenir un ensemble diversifié de solutions entières et réalisables.

Fischetti *et al.* [56] utilisent également deux vecteurs et une modification du modèle linéaire originel pour créer une solution entière réalisable. Notons toujours x^* la solution optimale de (LP), et notons maintenant $\tilde{x}$ le vecteur obtenu en arrondissant chaque variable de x^* à l'entier le plus proche. Pour un problème ne comportant que des variables binaires, l'heuristique de *feasibility pump* commence par remplacer la fonction objectif du problème originel par la distance :

$$\sum_{j=1}^{n} |x_j - \tilde{x}_j| = \sum_{j\in\{1,\ldots,n\},\tilde{x}_j=0} x_j + \sum_{j\in\{1,\ldots,n\},\tilde{x}_j=1} (1 - x_j)$$

Si la solution optimale x^* de ce problème linéaire modifié n'est pas entière, alors la solution de référence $\tilde{x}$ est modifiée en changeant la valeur de certaines variables de 0 à 1 et vice-versa suivant les valeurs prises dans le vecteur x^*, et un nouveau problème linéaire modifié est résolu. Cette heuristique crée ainsi deux suites $\tilde{x}$ (solutions entières) et x^* (solutions réalisables) dont on espère qu'elles sont convergentes et qu'elles ont la même limite qui sera ainsi à la fois entière et réalisable.

1.3.3 Heuristiques de réparation de solutions entières irréalisables

La deuxième idée naturelle qui est à la base de nombreuses heuristiques pour le *branch-and-cut* est de partir d'une solution entière irréalisable et d'explorer les solutions entières voisines avec un algorithme de recherche locale afin de restaurer la réalisabilité. Le voisinage d'un vecteur entier x est généralement constitué des vecteurs où l'on a modifié une ou deux composantes, en changeant la valeur de 0 à 1 et vice-versa dans le cas des variables binaires, et en retranchant ou ajoutant 1 (ou un entier inférieur à un maximum donné) dans le cas des variables entières générales. Le même type de voisinage peut être utilisé pour améliorer une solution entière réalisable [80, 10].

Abramson *et al.* [3] et Connolly [34] utilisent la métaheuristique du recuit simulé pour explorer l'espace des solutions entières. Pour guider la recherche vers une solution réalisable, Abramson et al. [3] présentent deux approches. La première pénalise l'irréalisabilité dans la fonction objectif. L'inconvénient de cette approche est que l'algorithme peut s'arrêter dans un optimum local qui n'est pas réalisable. Une approche plus efficace consiste à choisir la variable à changer en fonction de la violation dans la solution courante des contraintes où cette variable intervient, et de la réduction de ces violations qui pourrait être obtenue en changeant la valeur de cette variable. Walser [156] utilise une stratégie similaire : d'abord une contrainte violée est sélectionnée, puis une variable est sélectionnée pour diminuer la violation de cette contrainte. Ce choix de voisins est employé à l'intérieur d'une métaheuristique de marche aléatoire (Walksat [134]) et une liste de mouvements tabous est également mise en œuvre.

La solution de départ de ces algorithmes de recherche locale qui explorent l'espace des solutions entières est souvent obtenue par une heuristique d'arrondis, notamment une technique d'arrondi aléatoire [156].

1.3.4 Heuristiques d'énumération de points extrêmes

L'heuristique la plus étudiée et qui a donné lieu au plus grand nombre d'expérimentations est l'heuristique *pivot and complement* [10], que nous désignerons par P&C. L'idée de départ est que le programme linéaire en variables binaires (IP) défini à la page 37 est équivalent au problème linéaire suivant, où l'on a introduit les variables d'écart $s \in R^m_+$:

$$\begin{cases} \min\ cx \\ Ax - s = b \\ 0 \leq x_j \leq 1 \qquad \forall\ j \in \{1, \ldots,\ n\} \\ s \geq 0 \\ s_i \text{ est en base} \quad \forall\ i \in \{1, \ldots,\ m\} \end{cases}$$

En effet, si les m variables d'écart sont en base, alors les n variables x_j d'origine sont nécessairement hors base, donc prennent une valeur entière.

P&C part de la solution optimale de la relaxation linéaire de (IP) et procède par pivots successifs pour (i) augmenter le nombre de variables d'écart en base tout en

conservant la réalisabilité pour les contraintes linéaires, (ii) réduire la somme des défauts d'intégralité $\sum_{j=1}^{n} \min\{x_j, 1-x_j\}$ tout en conservant le même nombre de variables d'écart en base, ou bien (iii) introduire une variable d'écart en base quitte à produire une solution irréalisable pour certaines des contraintes linéaires. Certaines variables binaires sont également complémentées (x_j prend la valeur 1 si elle était à 0 et vice-versa) afin de réduire l'irréalisabilité pour les contraintes linéaires. Lorsque cette série de pivots et complémentations atteint une solution réalisable pour les contraintes linéaires et entière, alors P&C essaie de l'améliorer par une série de complémentations simples (une seule variable), doubles ou triples (deux ou trois variables sont complémentées simultanément) qui doivent conserver la réalisabilité.

P&C peut être intégrée dans des métaheuristiques telles que la recherche tabou si on la considère comme une boîte noire qui prend en entrée une solution de base pour un programme linéaire et donne en sortie (en cas de succès) une solution réalisable entière. Aboudi et Jörnsten [2] spécifient par exemple les attributs de la recherche tabou en modifiant le programme linéaire ou la solution de départ utilisés par l'heuristique P&C. Le critère d'aspiration est mis en œuvre par une coupe sur la valeur de la fonction objectif de façon à obtenir des solutions qui améliorent d'un certain facteur au moins la meilleure solution entière obtenue. Les caractéristiques taboues sont spécifiées par des coupes qui excluent les solutions entières préalablement obtenues. La diversification est obtenue en modifiant la fonction objectif du programme linéaire ou en appelant l'heuristique P&C sur une solution de base produite par *k* pivots arbitraires à partir de la solution optimale de la relaxation.

L'hybridation entre recherche tabou et pivots du simplexe a donné lieu à de nombreuses autres recherches [92, 67, 93, 100]. Une autre heuristique fondée sur les pivots du simplexe est *pivot, cut, and dive* [107] qui utilise de plus les coupes d'intersection [8] et les heuristiques de plongeon. Une autre heuristique d'énumération de points extrêmes est l'heuristique OCTANE [9] qui est aussi fondée sur les coupes d'intersection. Glover et Laguna proposent également un cadre général pour des heuristiques similaires [68, 69].

1.3.5 Vers des heuristiques intégrées à la recherche arborescente

Les heuristiques des trois classes que nous venons de présenter n'utilisent de fait que l'aspect linéaire de la programmation linéaire en nombres entiers (Sections 1.3.2 et 1.3.4) ou que son aspect combinatoire (Section 1.3.3). Pour exploiter ces deux aspects simultanément, il serait intéressant d'intégrer plus étroitement les heuristiques à l'exploration de l'arbre de *branch-and-cut*. Les heuristiques d'arrondis peuvent certes être appelées à partir de la relaxation de différents nœuds de l'arbre de *branch-and-cut*, mais aucune information globale sur cette population de relaxations continues n'est utilisée.

Au contraire, French *et al.* [61] proposent de combiner l'exploration arborescente du *branch-and-cut* avec un algorithme génétique. À chaque nœud de l'arbre de *branch-*

and-cut, la solution de la relaxation continue est arrondie, et si la solution entière (mais pas nécessairement réalisable) résultante est meilleure pour un certain critère d'adaptation (*fitness*) que la moyenne des individus considérés par l'algorithme génétique, alors elle est ajoutée à cette population. Si l'algorithme génétique produit par une succession de croisements et de mutations une nouvelle solution réalisable améliorante, alors celle-ci est transmise à l'algorithme de *branch-and-cut* et la recherche arborescente continue en créant et en explorant en priorité les nœuds qui correspondent aux fixations de variables entières réalisées dans cette solution. Ainsi, la population des relaxations continues est considérée de manière globale et le résultat de l'heuristique sert également à guider la recherche dans l'arbre de *branch-and-cut*. Le point faible de cette hybridation est le choix du critère d'adaptation pour l'algorithme génétique. S'il favorise la fonction objectif, alors l'algorithme génétique aura des difficultés à produire des solutions réalisables. Mais s'il favorise la réalisabilité, alors il sera difficile de produire des meilleures solutions que celles déjà obtenues par *branch-and-cut*.

La deuxième heuristique récente qui propose une intégration plus forte des techniques de recherche locale et de la recherche arborescente est le *local branching* [57]. Cette heuristique définit pour la première fois un voisinage de solutions entières différent et plus vaste que celui présenté à la Section 1.3.3. De manière comparable à l'idée d'exploration de grands voisinages présentée à la Section 1.2.6, ce voisinage est exploré par *branch-and-cut*. Cette heuristique permet d'obtenir des solutions de qualité sur un ensemble difficile de problèmes linéaires quelconques en nombres entiers. Dans le Chapitre 3, nous décrirons cet algorithme plus en détail et nous comparerons ses propriétés et ses performances avec notre propre heuristique de *large neighborhood search* pour des PLNE quelconques.

1.4 Conclusion : les deux objectifs de cette thèse

1.4.1 Un nouveau paradigme pour de meilleurs algorithmes hybrides

Nous avons montré dans la Section 1.1 que les algorithmes hybrides ont un fort potentiel mais qu'il existe encore de nombreuses difficultés qui empêchent de généraliser leur application. Le premier objectif de notre thèse est donc non seulement de proposer des algorithmes hybrides efficaces et robustes, mais aussi de réfléchir aux principes qui sous-tendent le développement des algorithmes hybrides, et de fournir des outils conceptuels pour faciliter leur généralisation. Nous avons déjà mis en évidence deux schémas d'hybridation, la décomposition et la recherche multiple, qui englobent la plupart des algorithmes hybrides existants. Nous avons également proposé dans la Section 1.2 une taxinomie plus détaillée des algorithmes hybrides où interviennent des techniques de recherche locale. Nous nous proposons dans les chapitres suivants de formaliser un nouveau paradigme d'hybridation qui permette de créer de meilleurs al-

gorithmes hybrides qui souffrent moins des inconvénients (coûts fixes et complexité logicielle accrus) dus à la coopération entre composants logiciels. Nous nous intéressons en particulier aux algorithmes hybrides *génériques*, c'est-à-dire qui peuvent non seulement s'appliquer avec succès à divers types de problèmes, mais qui peuvent surtout s'appliquer à tout type de problème modélisable dans un système donné, sans information extérieure sur la structure du problème à traiter.

Les algorithmes de définition et d'exploration de grands voisinages (Section 1.2.6) et l'heuristique de *local branching* (Section 1.3.5) nous semblent les précurseurs de cette nouvelle classe d'algorithmes hybrides.

1.4.2 Des heuristiques plus puissantes pour la programmation linéaire en nombres entiers

Nous avons montré dans la Section 1.3 que les heuristiques intégrées au *branch-and-cut* sont essentielles à la résolution de programmes linéaires en nombres entiers parce qu'elles sont complémentaires des stratégies de branchement. Cependant, comme l'ont mis en évidence les progrès réalisés par le *local branching* [57], la plupart des heuristiques existantes ne sont pas assez puissantes pour résoudre de manière efficace certains modèles difficiles. De plus, la plupart des heuristiques considèrent de fait uniquement l'aspect linéaire ou uniquement l'aspect combinatoire de la programmation linéaire en nombres entiers et n'exploitent pas les récents progrès réalisés dans la communauté des algorithmes hybrides. Enfin, la distinction claire en programmation linéaire en nombres entiers entre modélisation et résolution fait du *branch-and-cut* un bon candidat pour l'émergence d'algorithmes hybrides génériques. Nous nous proposons donc de tirer parti des différents travaux effectués sur les algorithmes hybrides pour mettre au point des heuristiques génériques et puissantes pour la programmation linéaire en nombres entiers.

Chapitre 2

Heuristiques pour le *branch-and-price*

Ce chapitre présente un schéma de coopération entre recherche locale et *branch-and-price* qui généralise au *branch-and-price* l'idée des heuristiques pour le *branch-and-cut*. Nous l'avons appliqué en particulier au problème de routage de véhicules avec fenêtres de temps. Ce travail a utilisé et a contribué à développer les outils logiciels de génération de colonnes et de résolution de plus court chemin contraint développés au sein de l'équipe « Solveurs coopératifs » d'ILOG. Ce chapitre reprend en grande partie le chapitre de livre à paraître [41].

2.1 Introduction

Comme nous l'avons déjà brièvement mentionné au chapitre précédent (p. 19 et p. 22), la génération de colonnes est une technique de résolution efficace pour traiter des problèmes combinatoires difficiles. Cette technique consiste à résoudre itérativement un problème maître (problème linéaire sur les colonnes déjà générées) et un sous-problème (qui génère des colonnes améliorantes). Dans le cas où le problème maître contient des variables entières, la génération de colonnes et une exploration arborescente de type *branch-and-bound* sont combinées pour former l'algorithme de *branch-and-price* [12]. De nombreuses stratégies peuvent être employées pour accélérer chacun des aspects du *branch-and-price*. Ces stratégies permettent souvent de réduire significativement le temps de résolution et certains problèmes ne pourraient pas être résolus sans les mettre en œuvre. Desaulniers *et al.* [49] passent en revue les stratégies d'accélération utilisées pour résoudre par *branch-and-price* des problèmes de routage de véhicules et d'affectation d'équipages et les classent en cinq catégories : stratégies de prétraitement, stratégies pour le sous-problème, stratégies pour le problème maître, stratégies pour l'exploration arborescente, et stratégies de postopti-

misation. Les techniques de recherche locale sont mentionnées trois fois dans cette classification : pour générer des solutions primales et duales initiales ; pour générer des solutions entières supplémentaires pour le problème maître en arrondissant les valeurs fractionnaires de la solution de sa relaxation continue à 1 ou à l'entier le plus proche ; pour postoptimiser la meilleure solution entière connue obtenue au terme de la limite de temps. Nous présentons ici un schéma général de coopération entre *branch-and-price* et recherche locale qui généralise ces trois accélérations. Il peut être également vu comme la généralisation au *branch-and-price* des heuristiques pour le *branch-and-cut* dont nous avons dressé un état de l'art au chapitre précédent.

Pour mesurer expérimentalement l'efficacité de ce schéma de coopération, nous l'avons appliqué au problème de routage de véhicules avec fenêtres de temps (*Vehicle Routing Problem with Time Windows* : VRPTW). De nombreux problèmes industriels sont des variantes du problème de routage de véhicules dont la formulation est la suivante. Étant donné un ensemble de clients qui demandent chacun une certaine quantité d'un bien, un ensemble de véhicules qui doivent partir d'un dépôt et revenir à la fin de leur tournée à ce même dépôt, et une matrice de distances entre chaque couple client-client et dépôt-client, il s'agit d'établir pour chaque véhicule la liste ordonnée des clients qu'il visitera de façon à minimiser la distance totale parcourue et parfois également le nombre de véhicules requis de façon à ce que la demande de chaque client soit satisfaite. Les contraintes additionnelles classiques sont les contraintes de capacité maximale pour les véhicules et les fenêtres de temps qui spécifient à quelle heure de la journée chaque client peut être desservi : l'ajout de ces deux contraintes définit le problème de routage de véhicules avec fenêtres de temps (VRPTW). Cordeau *et al.* [37] proposent une synthèse des différentes méthodes existantes pour résoudre ce problème. Parmi les méthodes exactes, le *branch-and-price* a récemment été appliqué avec succès à ce problème, comme en témoignent les travaux de Desrochers *et al.* [51], Kohl *et al.* [94], Larsen [96], Cook et Rich [35], Irnich [87], Chabrier *et al.* [27], Chabrier [26], et Irnich et Villeneuve [88]. La génération de colonnes fournit une borne inférieure (en cas de minimisation, comme nous le supposons dans toute la suite) sur le coût de la solution. Cette borne est améliorée au fur et à mesure de la progression dans l'arbre de *branch-and-price*. Cependant, il est parfois difficile pour le *branch-and-price* de générer des bonnes solutions rapidement car la solution du problème maître relâché est très souvent fractionnaire. Les techniques de recherche locale sont également souvent employées pour résoudre les problèmes de routage de véhicules avec fenêtres de temps, comme l'illustrent les travaux de Rochat et Taillard [124], Homberger et Gehring [82], Gambardella *et al.* [64], Cordeau *et al.* [38], De Backer *et al.* [46] et le logiciel ILOG DISPATCHER [86]. Étant donné une solution ou un ensemble de solutions [82, 64], des opérateurs qui définissent un voisinage sont appliqués itérativement pour se déplacer de solution en solution. Des métaheuristiques telles que la recherche locale guidée [46, 86] ou la recherche tabou [38, 86] sont utilisées pour alternativement concentrer la recherche sur un sous-espace intéressant (intensification) et s'échapper de

minima locaux pour explorer des régions différentes de l'espace de recherche (diversification). En général, les algorithmes de recherche locale produisent rapidement des solutions de très bonne qualité. Cependant, ce sont des algorithmes incomplets, ils ne fournissent pas de borne inférieure sur la fonction objectif. La différence entre la solution obtenue et la solution optimale ne peut donc pas être mesurée et l'utilisateur ne dispose pas d'informations pertinentes pour décider si plus de temps doit être alloué à l'algorithme dans l'espoir d'obtenir de meilleures solutions. Une manière simple d'obtenir à la fois des solutions de qualité en un temps réduit et une borne inférieure de qualité sur la fonction objectif serait d'invoquer en parallèle le *branch-and-price* et la recherche locale. Nous proposons ici un algorithme dans lequel le *branch-and-price* et la recherche locale coopèrent de manière plus étroite.

Nous décrivons notre schéma de coopération dans la Section 2.2. Puis nous détaillons dans la Section 2.3 comment nous l'avons appliqué au problème de routage de véhicules avec fenêtres de temps. Nous présentons et nous discutons nos résultats expérimentaux dans la Section 2.4. Enfin, nous résumons nos conclusions dans la Section 2.5.

2.2 Un schéma de coopération général entre *branch-and-price* et recherche locale

2.2.1 Algorithme

La Figure 2.1 présente notre schéma de coopération entre *branch-and-price* et recherche locale. La partie gauche de la figure représente les deux composants habituels d'un algorithme de génération de colonnes : le problème maître (relaxation linéaire) et le sous-problème. Sur la partie droite figurent deux techniques pour générer des solutions entières. Premièrement, le problème maître est résolu régulièrement sur l'ensemble courant de colonnes et sans relâcher les contraintes d'intégralité. Si ce programme linéaire en nombres entiers est résolu au nœud racine de l'arbre de *branch-and-price*, alors la meilleure solution entière connue à ce stade peut être utilisée comme solution de départ (*MIP start*) pour sa résolution par *branch-and-cut*. L'effort alloué à ce composant de PLNE est contrôlé par une limite en temps ou en nombre de nœuds imposée sur l'arbre de *branch-and-cut*. Quand cette limite est atteinte, l'exploration de l'arbre de *branch-and-price* reprend. Deuxièmement, un algorithme de recherche locale est également invoqué régulièrement. Il utilise comme solution initiale la meilleure solution réalisable connue. Contrairement au composant de programmation linéaire en nombres entiers, la recherche locale ne se limite pas à la combinaison des colonnes déjà générées, elle peut aussi introduire de nouvelles colonnes. Les colonnes générées sont ainsi plus diversifiées, ce qui est susceptible d'améliorer la convergence de la génération de colonnes à chaque nœud, en particulier en aidant à casser les phénomènes de dégénérescence.

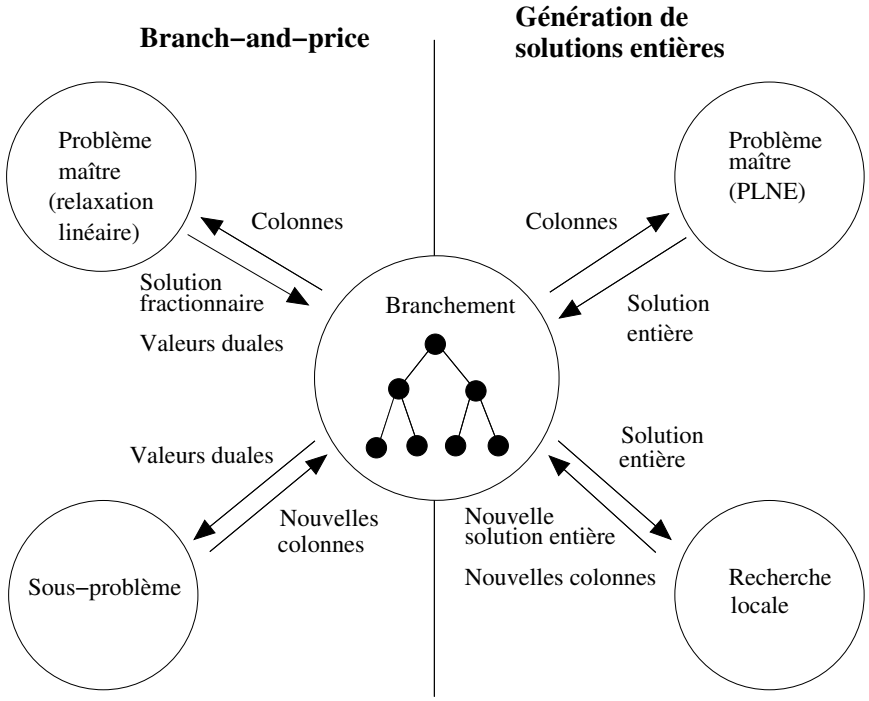

FIG. 2.1 – Notre schéma de coopération entre *branch-and-price* et recherche locale.

Cet algorithme hybride obéit au paradigme de recherche multiple que nous avons présenté à la Section 1.1.4 du chapitre précédent. Sa caractéristique principale est la diversification obtenue grâce à l'utilisation de deux algorithmes différents pour résoudre le même problème. Le *branch-and-price* bénéficie visiblement de la recherche locale qui est généralement plus efficace pour générer de bonnes solutions réalisables. Inversement, la recherche locale bénéficie du *branch-and-price* qui lui fournit des solutions initiales diversifiées. En effet, la principale difficulté des algorithmes de recherche locale est de réussir à s'échapper des minima locaux. Pour diversifier la recherche, les métaheuristiques tentent généralement de contrôler une suite de déplacements vers des solutions de moindre qualité afin d'atteindre une région différente et plus intéressante de l'espace de recherche. Il existe des schémas de diversification plus simples, par exemple la stratégie de démarrages multiples (*multi-start*) qui construit une nouvelle solution aussi éloignée que possible de l'optimum local courant afin d'explorer une région très différente de l'espace de recherche. Dans tous les cas, la diversification est obtenue aux dépens de l'optimisation de la fonction objectif. Au contraire, dans notre schéma de coopération, le composant de programmation linéaire en nombres entiers diversifie la recherche locale sans dégrader la fonction objectif. La borne supérieure pour le problème maître relâché et en nombres entiers est toujours mise à jour avec la valeur de la meilleure solution entière connue. Donc, lorsqu'une nouvelle solution

entière est découverte par le composant de PLNE ou parce que la solution du problème maître relâché est entière, elle est par construction de meilleur coût que le dernier optimum local trouvé par recherche locale. Ainsi, de la diversification est introduite dans la résolution, et dans le même temps, la fonction objectif est améliorée. Cette stratégie de diversification a néanmoins un certain coût : résoudre un programme linéaire en nombres entiers est plus coûteux en termes de temps de calcul que les schémas plus classiques de diversification.

Le *branch-and-price* étant une méthode complète, la solution optimale sera nécessairement trouvée si l'arbre de *branch-and-price* est exploré complètement. L'intérêt de notre schéma de coopération est de fournir des solutions entières de qualité plus rapidement que le *branch-and-price* pur. Il est donc utile de l'employer lorsque la recherche est tronquée, par exemple pour des instances qui ne peuvent pas être résolues actuellement à l'optimalité en un temps raisonnable.

2.2.2 Relation avec les algorithmes de la littérature

Notre schéma de coopération est une généralisation des trois stratégies d'accélération à base de recherche locale que nous avons mentionnées dans l'introduction. En effet, d'une part l'algorithme de recherche locale est invoqué régulièrement tout au long de l'exploration de l'arbre de *branch-and-price*, pas seulement au début ou à la fin du processus d'optimisation. Comme nous l'avons expliqué dans le paragraphe précédent, c'est ce qui permet aux deux techniques d'interagir de manière fructueuse pour générer des solutions entières de qualité plus rapidement. D'autre part, n'importe quel algorithme de recherche locale peut être employé dans notre schéma de coopération — pas seulement des techniques simples d'arrondis. Des heuristiques très efficaces et spécifiques au problème à résoudre peuvent être utilisées, comme nous le montrerons dans notre application au problème de routage de véhicules avec fenêtres de temps. Une approche similaire a été appliquée à un problème complexe de tournées de véhicules avec une flotte hétérogène et de nombreuses contraintes additionnelles : Savelsbergh et Sol [130] essaient de contruire une solution entière à partir de la solution du problème maître relâché, puis l'améliorent avec trois algorithmes simples de recherche locale spécifiques au problème traité.

Notre schéma de coopération est une généralisation au *branch-and-price* des heuristiques pour le *branch-and-cut* car la recherche locale permet ici non seulement de trouver de meilleures solutions entières, mais aussi d'introduire de nouvelles colonnes pour la formulation du problème maître. Il est à noter que la diversification rapide de l'ensemble des colonnes générées ainsi obtenue pourrait également être obtenue en résolvant le sous-problème avec plusieurs algorithmes différents, par exemple avec un algorithme heuristique comme dans les travaux de Savelsbergh et Sol [130], et Xu *et al.* [157], en plus de la résolution par programmation dynamique.

2.2.3 Choix des paramètres

Quelques expériences numériques sont souvent nécessaires pour déterminer quand et avec quelle limite de temps les composants de programmation linéaire en nombres entiers et de recherche locale doivent être invoqués. Cette paramétrisation dépend nécessairement du problème à résoudre mais nous pouvons isoler quelques règles valables en général. Il est coûteux de résoudre à l'optimalité le problème maître avec contraintes d'intégralité sur l'ensemble des colonnes déjà générées, donc il est nécessaire d'imposer une limite en temps ou en nombre de nœuds sur sa résolution par *branch-and-cut*. Il est souvent préférable d'invoquer le composant de PLNE dans les cas où il a de fortes chances de trouver une nouvelle solution entière, par exemple lorsque l'écart entre la valeur de la solution entière courante et la valeur du problème maître relâché est important, ou lorsque le nombre de variables fractionnaires dans la solution du problème maître relâché est faible. Le composant de recherche locale doit être appelé au moins chaque fois qu'une nouvelle solution entière est trouvée par le composant de PLNE ou parce que la solution du problème maître relâché est entière. Le composant de recherche locale peut être invoqué plus souvent, pour optimiser les solutions qu'il a lui-même trouvées, si la recherche locale s'avère beaucoup plus efficace que la programmation mathématique pour trouver des solutions réalisables. Il est facile de mettre en œuvre un schéma adaptatif qui augmente ou diminue la fréquence et le temps alloué aux composants de PLNE et de recherche locale suivant leur taux respectif de succès. Si l'on dispose d'un système multiprocesseur, il est également possible d'appeler en parallèle le composant de PLNE et le composant de recherche locale.

2.3 Application au problème de routage de véhicules avec fenêtres de temps

2.3.1 Le composant de *branch-and-price*

Nous avons utilisé le modèle classique (voir par exemple [37]) où chaque colonne correspond à une tournée réalisable. Soit $\{1,...,n\}$ l'ensemble des clients à desservir. Pour chaque tournée réalisable t, soit x_t la variable définie par :

$$x_t = \begin{cases} 1 \text{ si la tournée } t \text{ est utilisée dans la solution} \\ 0 \text{ sinon} \end{cases}$$

et notons c_t le coût de la tournée t. Le problème de routage de véhicules avec fenêtres de temps s'écrit maintenant sous la forme :

$$\begin{array}{ll} \min \sum_{t \in T} c_t x_t & \\ \sum_{t \in T} \delta_{it} x_t = 1, & \forall i \in \{1,...n\} \\ x_t \in \{0,1\}, & \forall t \in T \end{array}$$

où T est l'ensemble des tournées réalisables pour les contraintes de capacité des véhicules et les contraintes de fenêtres de temps, et $\delta_{it} = 1$ si le client i est visité pendant la tournée t, et vaut 0 sinon.

Décomposition en problème maître et sous-problème

La première difficulté de ce modèle vient du fait que le nombre de tournées réalisables augmente exponentiellement avec le nombre de clients. Nous utilisons donc la génération de colonnes pour générer des tournées intéressantes au fur et à mesure. Le modèle est décomposé en un problème maître et un sous-problème. Le problème maître est formulé de la manière suivante :

$$\min \sum_{t \in \hat{T}} c_t x_t \tag{2.1}$$

$$\sum_{t \in \hat{T}} \delta_{it} x_t = 1, \ \forall i \in \{1, ..., n\} \tag{2.2}$$

$$x_t \in \{0, 1\}, \ \forall t \in \hat{T} \tag{2.3}$$

où $\hat{T}$ est l'ensemble des tournées déjà générées. Nous résolvons en fait la relaxation continue du problème maître, en remplaçant la Contrainte 2.3 par la contrainte :

$$0 \leq x_t \leq 1, \forall t \in \hat{T} \tag{2.4}$$

Le sous-problème consiste à résoudre :

$$\min_{t \in T} c_t - \sum_{i=1}^{n} \pi_i \delta_{it}$$

où $(\pi_i)_{i \in \{1,...,n\}}$ est la valeur duale associée à l'Équation (2.2). Le sous-problème est à interpréter comme un problème de plus court chemin contraint sur le graphe d'origine où chaque arc $a = (i, j)$ est valué par son coût (sa distance) moins la valeur duale π_i associée à son extrémité initiale i.

Branch-and-price **et stratégie de branchement**

La deuxième difficulté de ce modèle concerne l'intégralité des variables x_t, comme indiqué dans la Contrainte 2.3. Le problème est donc résolu par *branch-and-price* :

1. Un ensemble initial de colonnes est généré, par exemple avec une heuristique simple.
2. La relaxation continue du problème maître est résolue, en remplaçant comme indiqué précédemment la Contrainte (2.3) par la Contrainte (2.4).
3. Le sous-problème est résolu avec les valeurs duales mises à jour à l'étape 2 et plusieurs chemins contraints de coût réduit négatif (s'il en existe) sont générés.
4. Les étapes 2 et 3 sont itérées jusqu'à ce que l'on ne puisse plus générer de tournées de coût réduit négatif.

5. Si la solution de la relaxation continue du problème maître est fractionnaire, alors la stratégie de branchement est appliquée et les étapes 2 et 3 sont itérées à chaque nœud de l'arbre ainsi construit.

Nous utilisons la règle de branchement suivante. Notons x^* la solution optimale du problème maître relâché après la dernière itération du sous-problème au nœud courant. Considérons le cas où le vecteur x^* n'est pas entier et notons $t_0 = \{i_0 = dépôt, i_1, i_2, \ldots, i_p, i_{p+1} = dépôt\}$ la tournée telle que $x^*_{t_0}$ est la variable la plus fractionnaire de x^*. $x_{t_0} < 1$, donc pour tout $k \in \{1, \ldots, p\}$, il existe d'autres tournées qui couvrent i_k et qui sont prises avec une valeur non-nulle dans la solution x^*. Pour toute tournée t avec $x^*_t > 0$ et qui a au moins un nœud en commun avec t_0, il existe $q \in \{1, \ldots, p\}$ tel que t couvre i_q, mais n'emprunte pas l'arc (i_q, i_{q+1}) ou n'emprunte pas l'arc (i_{q-1}, i_q). En effet, chaque tournée de $\hat{T}$ est unique, donc t et t_0 peuvent avoir une sous-séquence d'arcs commune, mais diffèrent nécessairement d'au moins un arc (dont l'extrémité initiale ou finale peut être le dépôt). Nous énumérons donc les colonnes déjà générées et nous choisissons la première tournée t telle que $x^*_t > 0$ et qui a au moins un nœud en commun avec t_0. Puis nous choisissons comme arc de branchement l'arc (i_q, i_{q+1}) pour le plus petit indice $q \in \{1, \ldots, p\}$ tel que $(i_q, i_{q+1}) \notin t$ (ou bien l'arc $(dépôt = i_0, i_1)$ si $(i_q, i_{q+1}) \in t$ pour tout $q \in \{1, \ldots, p\}$).

Sur l'une des branches, cet arc (i_q, i_{q+1}) est interdit. Sur l'autre branche, i_q et i_{q+1} peuvent être visités par une même tournée uniquement s'ils sont reliés par l'arc (i_q, i_{q+1}). En d'autres termes, sur la deuxième branche, tous les arcs (i_q, r) avec $r \neq i_{q+1}$ et (s, i_{q+1}) avec $s \neq i_q$ sont interdits. Cette stratégie de branchement est très pratique parce qu'elle est facile à mettre en œuvre dans le problème maître et dans le sous-problème.

Résolution du sous-problème

Le sous-problème est résolu par programmation dynamique, avec une adaptation de l'algorithme à base d'étiquettes de Desrochers [50] afin de résoudre le problème de plus court chemin contraint *élémentaire*. Le détail de cet algorithme a été présenté dans un article écrit avec Alain Chabrier et Claude Le Pape [27] et dans un article ultérieur d'Alain Chabrier [26]. La même idée a été développée indépendamment par Guéguen *et al.* [75]. Pourquoi générer uniquement des chemins élémentaires dans le sous-problème ? Si la fonction qui associe un coût à chaque arc respecte l'inégalité triangulaire, alors la solution optimale ne contiendra que des tournées élémentaires, que le sous-problème génère des tournées avec ou sans cycle. Il est plus facile de résoudre le problème du plus court chemin contraint sans la contrainte d'élémentarité, donc la plupart des modèles de génération de colonnes de la littérature autorisent le sous-problème à générer des tournées avec cycles et utilisent parfois concomitamment un mécanisme pour éliminer une partie des tournées avec cycles ou pour améliorer la borne inférieure sur la fonction objectif [95, 94, 35, 87, 88]. Ces stratégies sont essentielles pour résoudre des instances avec de grandes fenêtres de temps ou un horizon

long. En effet, sur ces instances, les données numériques du problème ne sont pas suffisamment contraignantes pour éliminer spontanément une partie des tournées avec cycles. Un cycle peut même être parcouru plusieurs fois par la même tournée.

En résumé, l'algorithme à base d'étiquettes utilisé pour résoudre le sous-problème de plus court chemin élémentaire fonctionne de la manière suivante. Des chemins partiels commençant au dépôt et visitant un certain nombre de clients sont construits. Comme toujours en programmation dynamique, les chemins partiels dominés sont éliminés au fur et à mesure : si le chemin partiel cp_1 et le chemin partiel cp_2 se terminent tous les deux par la visite du client i, mais si cp_1 arrive plus tôt en i, correspond à un véhicule moins chargé et est moins coûteux que cp_2, alors cp_2 est éliminé. En effet, pour chaque extension de cp_2 en un chemin complet, cp_1 pourrait être étendu exactement de la même façon et son extension serait moins coûteuse que l'extension de cp_2. Cependant, cette règle de dominance n'est plus valide si nous souhaitons calculer uniquement des chemins élémentaires. En effet, si l'extension de cp_2 dessert certains clients qui étaient déjà visités par cp_1 avant le client i, alors cp_1 ne peut être étendu de la même façon que cp_2, sinon cp_1 visiterait ces clients-là deux fois. Nous modifions donc la règle de dominance de la manière suivante : si le chemin partiel cp_1 et le chemin partiel cp_2 se terminent tous les deux par la visite du client i, mais si cp_1 arrive plus tôt en i, correspond à un véhicule moins chargé et est moins coûteux que cp_2, et si l'ensemble des clients visités par cp_1 est un sous-ensemble des clients visités par cp_2, alors le chemin partiel cp_2 est éliminé. Des raffinements de cette règle de dominance sont présentés en détail dans les deux articles déjà mentionnés [27, 26]. Nous résolvons d'abord les premières itérations du sous-problème avec la règle de dominance d'origine. Cette version est une heuristique pour le problème du plus court chemin contraint élémentaire puisqu'elle élimine des chemins qui ne sont pas dominés. Puis, lorsqu'il n'est plus possible de générer des chemins de coût réduit négatif, nous utilisons la règle de dominance modifiée qui résout de manière exacte le problème du plus court chemin contraint élémentaire.

Stratégies d'accélérations

Nous avons utilisé un certain nombre de stratégies bien connues pour accélérer la résolution de notre modèle de *branch-and-price*, en particulier dans le sous-problème. Elles sont énumérées dans [27, 26]. En ce qui concerne le problème maître, l'Équation 2.2 de partitionnement est remplacée par l'équation suivante de couverture :

$$\sum_{t \in \hat{T}} \delta_{it} x_t \geq 1, \ \forall i \in \{1, \ldots, n\} \tag{2.5}$$

Les solutions entières peuvent donc visiter certains clients plus d'une fois, en particulier au début du processus de *branch-and-price*. Pour chaque solution trouvée par le composant de PLNE ou parce que la solution du problème maître relâché est spontanément entière, nous appliquons une heuristique gloutonne qui enlève itérativement

chaque client desservi plus d'une fois de toutes les tournées sauf de la tournée où son retrait entraînerait la réduction de coût la plus faible. Cette heuristique simple nous permet d'améliorer la solution entière courante et les colonnes résultantes sont également ajoutées à l'ensemble des colonnes déjà générées.

2.3.2 Les heuristiques

Construction d'une première solution

Les heuristiques sont utilisées dans deux contextes différents. Une heuristique est d'abord utilisée pour construire une première solution. Cette solution initiale peut être aussi simple que la solution triviale « un client par tournée ». Dans notre schéma de coopération, et dans le schéma de recherche locale pure, la solution initiale est construite par l'heuristique *savings* [32] adaptée au problème avec fenêtres de temps.

Amélioration d'une solution existante

Un algorithme de recherche locale est ensuite employé pour améliorer une solution existante. Nous avons utilisé deux algorithmes, le premier est relativement simple et le deuxième est plus sophistiqué. Les expériences que nous avons conduites avec ces deux exemples différents de recherche locale et qui sont relatées dans la Section 2.4 nous permettront de montrer que notre schéma de coopération est efficace dans des situations variées. Ces expériences montreront aussi que même un algorithme simple de recherche locale permet d'améliorer les capacités du *branch-and-price* à trouver rapidement des solutions entières de bonne qualité.

Large Neighborhood Search

Nous avons d'abord implémenté un algorithme de *large neighborhood search* (LNS) très proche de celui de Shaw [135] que nous avons résumé à la Section 1.2.6 du chapitre précédent. Cet algorithme est facile à implémenter avec ILOG SOLVER.

Cet algorithme de grands voisinages s'est révélé trop lent pour des voisinages correspondant à la remise en cause de la position de plus de 20 clients. Pour notre algorithme de recherche locale pure, nous avons donc mis en œuvre une stratégie de redémarrages multiples pour augmenter la diversification. Quand le voisinage atteint une taille de 20 clients à repositionner, nous construisons avec une heuristique d'insertion une solution assez éloignée de la solution courante (et souvent de coût beaucoup plus élevé) de la manière suivante. Les clients sont insérés dans l'ordre « orthogonal » de l'ordre de la solution courante : d'abord le premier client de chaque tournée est inséré, puis le deuxième client de chaque tournée, etc. Chaque client est inséré dans la position qui induit le coût le plus faible et des tournées supplémentaires sont ouvertes au fur et à mesure. À la fin de cette procédure d'insertion, les positions d'un certain

nombre de clients sont de plus modifiées aléatoirement. La solution ainsi obtenue sert de solution initiale pour une nouvelle phase de LNS.

ILOG DISPATCHER : *Guided Tabu Search*

Nous avons ensuite décidé d'utiliser un code très efficace de recherche locale pour les problèmes de routage de véhicules : ILOG DISPATCHER [86]. Les voisinages exploités par cet algorithme sont les voisinages 2-OPT, Or-OPT (similaire à 3-OPT), *Relocate* (un client est inséré dans une tournée différente), *Exchange* (échange de deux clients desservis par deux tournées différentes) et un voisinage similaire au voisinage *Cross* [142] (les parties finales de deux tournées sont échangées). La métaheuristique utilisée ici est la recherche tabou guidée (*Guided Tabu Search*) qui est un hybride entre la recherche locale guidée (GLS : *Guided Local Search*) et la recherche tabou. L'implémentation est décrite par De Backer et al. [46] et dans le manuel ILOG DISPATCHER [86].

La métaheuristique GLS [155] a pour but d'aider les algorithmes de descente à s'échapper des minima locaux. Elle consiste à optimiser une fonction objectif modifiée qui est la somme pondérée de la fonction objectif du problème d'origine et de pénalités correspondant à certaines caractéristiques des solutions déjà rencontrées. À chaque itération, cet objectif modifié est d'abord optimisé par l'algorithme de descente. C'est la phase d'intensification puisque l'objectif modifié fait intervenir (avec un certain coefficient de pondération) la fonction objectif d'origine. Puis, cet objectif est à nouveau modifié, en augmentant ou en réduisant les pénalités associées à certaines caractéristiques, suivant leur coût et le nombre d'itérations pendant lesquelles elles ont été pénalisées. Ce mécanisme de mémoire à long terme permet à l'algorithme de descente d'explorer différentes régions de l'espace de recherche.

Cependant, dans notre schéma de coopération, la diversification à long terme est déjà assurée par les solutions entières trouvées par le composant de programmation linéaire en nombres entiers ou lorsque la solution du problème maître relâché est spontanément entière. Il est néanmoins nécessaire d'assurer la diversification à court terme, et c'est le rôle de la recherche tabou. La recherche tabou [70] est une métaheuristique efficace et bien étudiée. En résumé, elle permet de s'échapper d'un minimum local en interdisant pendant un certain nombre d'itérations de remettre en cause les changements de valeur des variables les plus récemment modifiées, sauf si les changements inverses vérifient un certain critère d'aspiration, par exemple s'ils améliorent la meilleure solution connue. Si la longueur de la liste tabou est réduite, alors c'est un mécanisme de mémoire à court terme.

De Backer *et al.* [46] ont montré que, tels que ces algorithmes ont été implémentés dans ILOG DISPATCHER pour les problèmes de routage de véhicules, la recherche tabou guidée est plus efficace que la recherche locale guidée employée seule et que la recherche tabou employée seule.

2.4 Résultats expérimentaux

2.4.1 Instances étudiées

Toutes les expériences numériques décrites dans les prochaines sections ont été conduites sur les instances bien connues pour le problème de routage de véhicules avec fenêtres de temps qui ont été introduites par Solomon [136] et sur lesquelles de nombreux algorithmes exacts ou heuristiques ont été testés depuis. Nous avons adopté la convention utilisée par la plupart des méthodes exactes : l'objectif est de minimiser la distance totale parcourue sans se préoccuper du nombre de véhicules ; les distances et les temps de parcours sont déterminés par la distance euclidienne arrondie au dixième inférieur. Le jeu de données de Solomon comprend deux séries d'instances : dans la série 1, la capacité des véhicules est limitée et l'horizon des problèmes est assez court ; dans la série 2 au contraire, les véhicules ont une plus grande capacité et l'horizon est plus long, ce qui permet de desservir un plus grand nombre de clients par tournée. Les instances de la série 1 sont donc plus faciles à résoudre parce que moins combinatoires : le nombre total de routes réalisables pour ces instances est moins élevé que pour les instances de la série 2. La littérature s'est focalisée jusqu'à très récemment sur la série 1, avec quelques exceptions notables [96, 35, 90, 87, 88, 27, 26]. Le lecteur est invité à consulter la revue de Cordeau *et al.* [37] pour un aperçu plus détaillé de la littérature. Les instances de Solomon sont de plus divisées en trois groupes. Pour les instances « R », les coordonnées géographiques des clients sont distribuées aléatoirement. Pour les instances « C », les clients sont repartis entre différents groupes compacts géographiquement. Pour les instances « RC », une partie des clients est distribuée aléatoirement alors que le reste est organisé en groupes. Chaque instance est un problème à 100 clients, à partir duquel sont construits des problèmes plus petits en ne prenant en compte que les 50 premiers clients.

Les Tableaux 2.1 et 2.2 énumèrent les valeurs avec lesquelles nous comparerons nos résultats dans les Sections 2.4.3 et 2.4.5. Ces valeurs de référence sont les meilleurs résultats dont nous avons connaissance, qui ont été rapportés dans la littérature ou que nous avons découverts dans nos propres expériences (nos algorithmes sont décrits plus en détail dans la section suivante), comme indiqué dans la colonne « Origine ». Les solutions marquées d'une astérisque ont été prouvées optimales. Lorsque l'optimum n'est pas connu, nous notons la meilleure borne inférieure connue (LB en italique) et la meilleure borne supérieure connue (UB). Le nombre de véhicules correspondant à chaque borne supérieure est indiqué entre parenthèses. Pour certaines instances ouvertes de la série 2, la génération de tournées élémentaires ne converge pas au nœud racine. Nous donnons alors la borne inférieure produite au nœud racine en autorisant la génération de tournées avec cycles (LBcycles), pour les instances où le *pricing* au nœud racine termine en un temps raisonnable. Dans toute la suite, nous mesurerons la qualité des solutions réalisables et des bornes inférieures obtenues par rapport à ces valeurs de référence. Pour la série 1, nous avons choisi comme valeur de référence la meilleure

borne inférieure connue. Comme aucune borne inférieure (autre que zéro) n'est connue pour certaines instance de la série 2, nous utilisons comme valeur de référence sur cette deuxième série le coût de la meilleure solution réalisable connue.

2.4.2 Méthodes

Rappelons que les méthodes que nous allons comparer dans les trois sections suivantes sont :

1. Notre schéma de coopération entre le *branch-and-price* et un des algorithmes de recherche locale précédemment décrits (BP+LNS, BP+DISPATCHER). Le composant de programmation linéaire en nombres entiers est appelé toutes les 4 minutes pour une durée de 1 minute. L'algorithme de recherche locale est appelé pour 10 secondes toutes les 2 minutes, et chaque fois qu'une nouvelle solution entière est trouvée par les autres composants. Si l'algorithme de recherche locale trouve une solution pendant l'une de ses invocations, il est immédiatement invoqué à nouveau pour 10 secondes. Ce mécanisme adaptatif très simple nous permet d'appeler l'algorithme de recherche locale plus souvent en cas de succès, sans trop ralentir la preuve d'optimalité après que la solution optimale est trouvée.
2. Quasiment le même algorithme de *branch-and-price* que dans notre schéma hybride, mais utilisé sans recherche locale. Les quelques différences mineures avec le *branch-and-price* et le composant de PLNE utilisés dans la coopération sont les suivantes :
   – Le composant de PLNE est invoqué plus souvent (toutes les 3 minutes au lieu de toutes les 4 minutes) avec la même limite de temps pour chaque appel (1 minute), afin de compenser l'absence de composant supplémentaire pour générer des solutions entières.
   – Pour la méthode « BP 1 », l'ensemble initial de colonnes est constitué par la solution triviale qui comprend un client par tournée. Pour la méthode « BP 2 », la solution initiale est construite par l'heuristique de *savings*, comme dans le cas du schéma de coopération.
3. Le même algorithme de recherche locale que dans la coopération (LNS ou ILOG DISPATCHER), mais utilisé seul. Pour LNS, nous utilisons en plus le schéma de démarrages multiples décrit en page 52.

Les paramètres de chaque algorithme ont été choisis expérimentalement. Il s'est avéré plus efficace d'invoquer le composant de PLNE souvent et avec une limite de temps courte, plutôt que plus rarement et avec une limite de temps plus grande, car nous avons mis en évidence expérimentalement que l'échec du composant de PLNE à générer des solutions entières améliorantes vient généralement de l'absence de colonnes améliorantes, plutôt que de l'incapacité de la programmation linéaire en nombres entiers à découvrir une combinaison améliorante des colonnes existantes.

Instance	50 clients		100 clients	
	Coût	Origine	Coût	Origine
C101	362.4 (5)*	[94]	827.3 (10)*	[94]
C102	361.4 (5)*	[94]	827.3 (10)*	[94]
C103	361.4 (5)*	[94]	826.3 (10)*	[94]
C104	358.0 (5)*	[94]	822.9 (10)*	[94]
C105	362.4 (5)*	[94]	827.3 (10)*	[94]
C106	362.4 (5)*	[94]	827.3 (10)*	[94]
C107	362.4 (5)*	[94]	827.3 (10)*	[94]
C108	362.4 (5)*	[94]	827.3 (10)*	[94]
C109	362.4 (5)*	[94]	827.3 (10)*	[94]
R101	1044.0 (12)*	[94]	1637.7 (20)*	[94]
R102	909.0 (11)*	[94]	1466.6 (18)*	[94]
R103	772.9 (9)*	[94]	1208.7 (14)*	[94]
R104	625.4 (6)*	[94]	971.5 (11)*	[88]
R105	899.3 (9)*	[94]	1355.3 (15)*	[94]
R106	793.0 (5)*	[94]	1234.6 (13)*	[35, 90]
R107	711.1 (7)*	[94]	1064.6 (11)*	[35, 90]
R108	617.7 (6)*	[94]	*LB = 919.9*, UB = 939	*[88]*, [62]
R109	786.8 (8)*	[94]	1146.9 (13)*	[35, 90]
R110	697.0 (7)*	[94]	1068 (12)*	[35, 90]
R111	707.2 (7)*	[35, 90]	1048.7 (12)*	[35, 90]
R112	630.2 (6)*	[35, 90]	*LB = 935.1* , UB = 960.1 (10)	*[35]*, ILOG DISPATCHER
RC101	944.0 (8)*	[94]	1619.8 (15)*	[94]
RC102	822.5 (7)*	[94]	1457.4 (14)*	[35, 90]
RC103	710.9 (6)*	[94]	1258 (11)*	[35, 90]
RC104	545.8 (5)*	[94]	1132.3 (10)*	[88]
RC105	855.3 (8)*	[94]	1513.7 (15)*	[94]
RC106	723.2 (6)*	[94]	*LB = 1356.1* , UB = 1376.4	*[35]*, [62]
RC107	642.7 (6)*	[94]	1207.8 (12)*	[88]
RC108	598.1 (6)*	[94]	1114.2 (11)*	[88]

TAB. 2.1 – Meilleures bornes inférieures et supérieures connues pour la série 1.

Instance	50 clients		100 clients	
	Coût	Origine	Coût	Origine
C201	360.2 (3)*	[35], [96]	589.1 (3)*	[35, 90]
C202	360.2 (3)*	[35], [90]	589.1 (3)*	[35, 90]
C203	359.8 (3)*	[35], [90]	588.7 (3)*	[90]
C204	350.1 (2)*	[90]	588.1 (3)*	[88]
C205	359.8 (3)*	[35], [90]	586.4 (3)*	[35, 90]
C206	359.8 (3)*	[35], [90]	586.0 (3)*	[35, 90]
C207	359.6 (3)*	[35], [90]	585.8 (3)*	[35, 90]
C208	350.5 (2)*	[35], [90]	585.8 (3)*	[90]
R201	791.9 (6)*	[35, 90]	1143.2 (8)*	[90]
R202	698.5 (5)*	[35, 90]	*LB* = 933.5, UB = 1029.6 (8)	*LBcycles*, BP+DISPATCHER
R203	605.3 (5)*	[87, 27, 26]	*LB* = *847.2*, UB = 871.4 (6)	*[88]*, ILOG DISPATCHER
R204	506.4 (2)*	[88]	LB = 0, UB = 733.0 (5)	[116]
R205	690.1 (5)*	[96, 90, 88]	*LB* = 932.6, UB = 951.9 (5)	*[88]*, [116]
R206	632.4 (4)*	[87, 27, 26]	*LB* = *840.7*, UB = 880.6 (4)	*[88]*, [116]
R207	*LB* = *560.5* , UB = 575.5 (3)	*[88]*, ILOG DISPATCHER, [116]	*LB* = *668.7*, UB = 794.0 (4)	*LBcycles*, [116]
R208	*LB* = *475.4* , UB = 487.7 (2)	*[88]*, ILOG DISPATCHER, [116]	LB = 0, UB = 701.2 (3)	[116]
R209	600.6 (4)*	[87, 27, 26]	*LB* = 835.9, UB = 855.7 (5)	*[88]*, [116]
R210	645.6 (4)*	[87, 27, 26]	*LB* = *855.7*, UB = 900.8 (6)	*[88]*, [116]
R211	535.5 (3)*	BP, BP+DISPATCHER, [88]	*LB* = *710.8*, UB = 751.7 (3)	*[88]*, [116]
RC201	684.8 (5)*	[96, 90]	1261.8 (9)*	[90]
RC202	613.6 (5)*	[87, 27, 26]	1092.3 (8)*	[27, 26]
RC203	555.3 (4)*	[27, 26, 88]	*LB* = 693.6, UB = 923.7 (5)	*LBcycles*, [116]
RC204	444.2 (3)*	BP+DISPATCHER	*LB* = *0*, UB = 783.5 (4)	[116]
RC205	630.2 (5)*	[87, 27, 26]	UB = 1154.0 (7)*	[27, 26]
RC206	610.0 (5)*	[87, 27, 26]	*LB* = *1018*, UB = 1051.1 (6)	*[88]*, [27]
RC207	558.6 (4)*	[27, 26]	*LB* = *892*, UB = 966.3 (5)	*[88]*, [27]
RC208	*LB* = *424.9* , UB = 476.7 (3)	*[88]*, [27]	*LB* = *0*, UB = *777.3* (3)	[116]

TAB. 2.2 – Meilleures bornes inférieures et supérieures connues pour la série 2.

Tous les résultats ont été obtenus avec une limite d'une heure pour chaque instance et chaque algorithme, sur un Pentium IV cadencé à 1.5 GHz avec le système d'exploitation Linux et 256 Mo de mémoire vive, avec les logiciels ILOG CPLEX 8.1.0, ILOG SOLVER 5.3 et ILOG DISPATCHER 3.3.

2.4.3 Qualité des solutions entières

Nous présentons maintenant les résultats principaux pour les méthodes que nous venons de décrire. Le Tableau 2.3 donne la qualité des solutions réalisables obtenues par chacun des algorithmes sur chaque série de Solomon, pour les instances de taille 50 et de taille 100. La qualité des solutions obtenues par chaque algorithme est mesurée par la déviation moyenne relative (en %) entre la borne supérieure obtenue par cet algorithme au terme de la limite de temps et les valeurs de référence énumérées dans les Tableaux 2.1 et 2.2. Il est à rappeler que, pour la série 2, les valeurs de référence sont les coûts de solutions réalisables potentiellement sous-optimales, donc les chiffres donnés sur cette série ne sont pas nécessairement un majorant de l'écart à la solution optimale. Les Figures 2.2 à 2.4 montrent l'évolution au cours du temps de la qualité des solutions pour les instances de taille 100, série par série. Le Tableau 2.4 résume enfin les résultats en agrégeant les chiffres précédents sur toutes les séries. La qualité des solutions est calculée comme dans le Tableau 2.3 et nous donnons aussi le nombre d'instances pour lesquelles chaque algorithme atteint (et entre parenthèses, prouve) l'optimum. Il est à noter que seuls le *branch-and-price* pur et notre schéma de coopération entre *branch-and-price* et recherche locale fournissent des bornes inférieures sur la fonction objectif, donc sont les seuls algorithmes à même de fournir une preuve d'optimalité. À des fins de comparaison, mentionnons pour finir que le jeu de données de Solomon comprend 56 instances de taille 50 et 56 instances de taille 100. La solution optimale est connue pour 53 des instances de taille 50 et 38 des instances de taille 100.

Algorithme	Nombre de clients	C1	R1	RC1	C2	R2	RC2
BP 1	50	0.00	0.15	1.15	0.74	3.73	2.05
	100	0.00	2.40	6.17	6.50	8.63	5.26
BP 2	50	0.07	0.25	1.32	0.68	3.12	2.50
	100	0.29	2.37	6.05	7.80	7.76	5.60
LNS pur	50	0.00	0.11	0.20	0.00	1.69	1.25
	100	0.00	2.96	3.93	2.61	5.85	7.18
DISPATCHER pur	50	0.00	0.03	0.12	0.40	0.89	1.13
	100	0.09	1.06	2.36	0.00	0.62	1.66
BP+LNS	50	0.00	0.00	0.25	0.12	3.27	0.38
	100	0.00	1.70	3.62	3.04	6.08	3.93
BP+DISPATCHER	50	0.00	0.12	0.51	0.75	1.15	0.36
	100	0.00	1.47	4.46	0.26	4.13	2.56

TAB. 2.3 – Qualité des solutions obtenues en une heure, série par série.

Les premières conclusions que nous pouvons tirer de cet ensemble de résultats sont les suivantes. Il est toujours plus efficace de combiner le *branch-and-price* et la re-

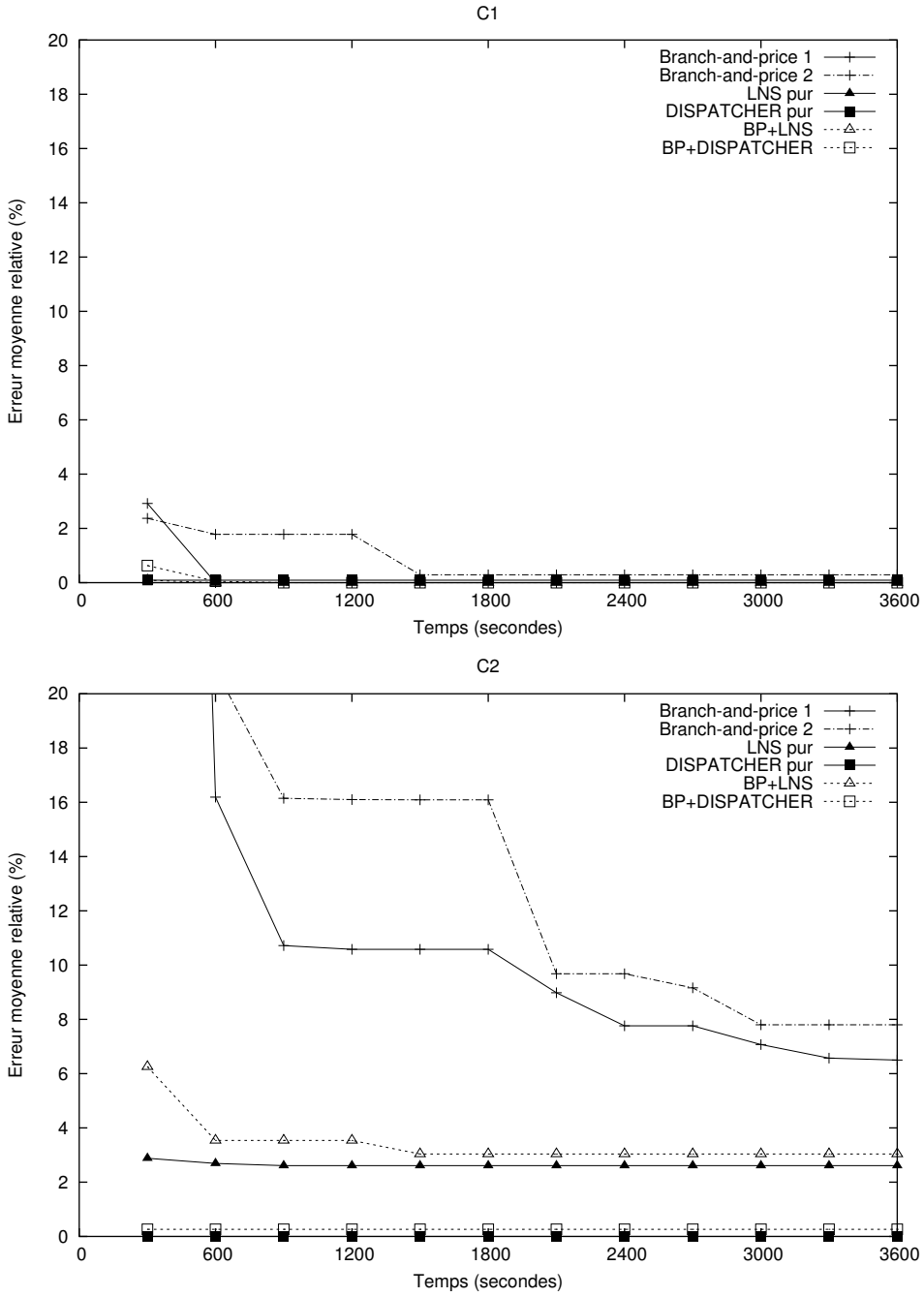

FIG. 2.2 – Évolution de qualité des solutions en fonction du temps pour la série C (instances de taille 100).

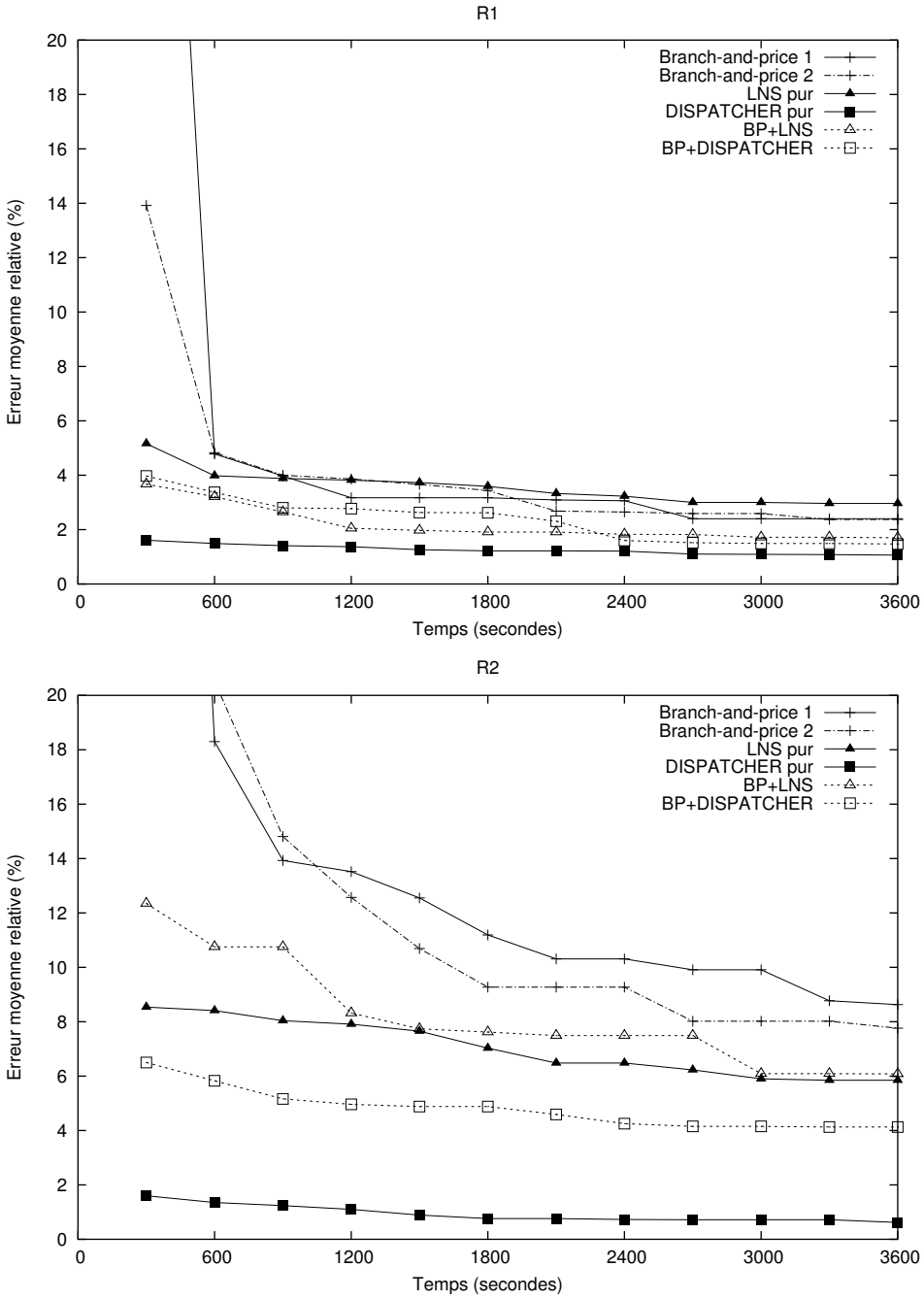

FIG. 2.3 – Évolution de qualité des solutions en fonction du temps pour la série R (instances de taille 100).

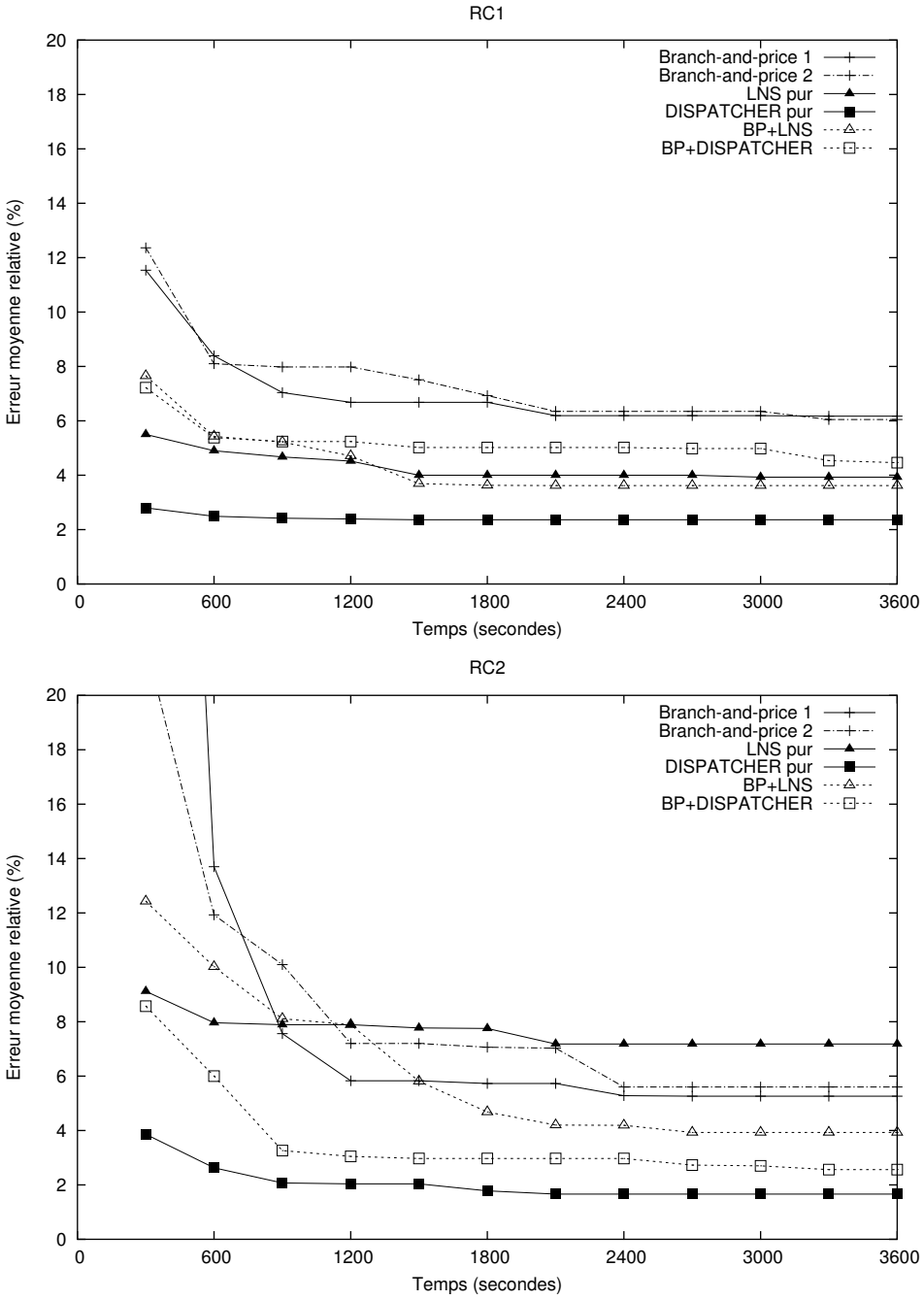

FIG. 2.4 – Évolution de qualité des solutions en fonction du temps pour la série RC (instances de taille 100).

Algorithme	Nombre de clients	Qualité des solutions	Nombre d'instances où l'optimum est atteint (et prouvé)
BP 1	50	1.33	40 (37)
	100	4.69	17 (15)
BP 2	50	1.32	41 (37)
	100	4.78	17 (15)
LNS pur	50	0.56	41 (-)
	100	3.66	11 (-)
DISPATCHER pur	50	0.42	32 (-)
	100	0.87	12 (-)
BP+LNS	50	0.75	47 (34)
	100	3.00	19 (15)
BP+DISPATCHER	50	0.48	45 (38)
	100	2.09	19 (16)

TAB. 2.4 – Résumé des résultats sur l'ensemble des instances de Solomon.

cherche locale selon notre schéma de coopération que d'utiliser le *branch-and-price* seul pour obtenir rapidement de bonnes solutions réalisables. Sur toutes les séries, quelle que soit la taille des instances considérées, nos deux algorithmes hybrides combinant *branch-and-price* et recherche locale (LNS ou ILOG DISPATCHER) sont aussi efficaces ou, dans la grande majorité des cas, plus efficaces que le *branch-and-price* pur. L'amélioration de performance constatée pour la coopération est le plus souvent corrélée à la performance de l'algorithme de recherche locale quand il est utilisé seul. Notre algorithme simple de LNS est logiquement moins efficace que l'algorithme plus sophistiqué de recherche tabou guidée implémenté dans ILOG DISPATCHER. De la même manière, la coopération entre *branch-and-price* et LNS est le plus souvent dominée par la coopération entre *branch-and-price* et ILOG DISPATCHER. Cependant, même un algorithme simple comme le LNS permet d'améliorer notablement les performances du *branch-and-price*.

Au contraire, en ce qui concerne les algorithmes de pur *branch-and-price*, le fait de partir d'une solution initiale construite par l'heuristique *savings* (« BP 2 ») ne permet pas d'améliorer significativement les performances du même algorithme qui part d'une solution initiale triviale (« BP 1 »). Notre tentative d'explication est que la simple heuristique de *savings* n'est pas assez puissante pour entraîner une différence significative de performance.

Revenons maintenant plus en détail sur quelques instances de taille 100 pour des remarques plus spécifiques. Sur les séries R1, RC1 et en particulier RC2, notre schéma de coopération entre *branch-and-price* et LNS domine à la fois l'algorithme de LNS pur et les deux variantes du *branch-and-price* pur. Ce phénomène illustre l'interaction fructueuse entre les deux composants de notre hybridation, chacun optimisant les solutions trouvées par l'autre composant. Sur la série RC2, le *branch-and-price* est plus efficace que le LNS pur et notre coopération est néanmoins à son tour plus efficace que ces deux algorithmes purs. Ceci montre la robustesse de notre schéma de coopération qui est toujours proche des performances de la méthode qui donne les meilleurs

résultats.

Les Figures 2.2 à 2.4 montrent que nos algorithmes hybrides donnent parfois en une heure des résultats inférieurs aux résultats obtenus en une demi-heure par recherche locale pure. En particulier, ILOG DISPATCHER permet de trouver très rapidement d'excellentes solutions sur toutes les séries, il est donc difficile de faire mieux. L'intérêt de notre schéma de coopération peut donc être remis en question, étant donné que des schémas de coopération plus simples aboutiraient parfois à de meilleurs résultats, par exemple en partageant naïvement le temps entre le *branch-and-price* et la recherche locale, ou en invoquant d'abord l'algorithme de recherche locale pour la moitié du temps alloué, puis en appelant le *branch-and-price* à partir de la solution obtenue par recherche locale. Mais la caractéristique essentielle de notre schéma de coopération est que, à chaque instant du processus d'optimisation, l'utilisateur connaît une borne inférieure et une borne supérieure sur la fonction objectif. L'utilisateur peut donc évaluer à chaque instant la qualité des solutions obtenues et arrêter l'optimisation dès qu'il est satisfait.

2.4.4 Quel composant trouve les solutions entières dans la coopération ?

Après avoir discuté des performances générales de notre schéma de coopération, nous présentons maintenant des résultats plus détaillés pour chacun de ses composants afin de mieux comprendre pourquoi notre schéma de coopération arrive à générer de bonnes solutions rapidement. Les tableaux 2.5 et 2.6 donnent pour chaque composant de la coopération le nombre de fois où il trouve une nouvelle solution améliorante divisé par le nombre de fois où ce composant est appelé (colonne *%s* comme succès), ou divisé par le nombre total de solutions entières trouvées par tous les composants (colonne *%c* comme contribution). Ces statistiques sont agrégées pour les instances de taille 100 des séries 1 et 2.

Composant	Série 1		Série 2	
	%s	%c	%s	%c
Problème maître relâché	0.01	4.65	0.01	1.85
PLNE	5.92	7.90	7.96	7.40
Heuristique visites multiples	62.96	7.90	80.00	7.40
Total LNS	27.22	79.53	29.03	83.33
... à partir d'une solution trouvée par :				
Branch-and-price	85.00	23.72	64.70	20.37
LNS	21.12	55.81	24.63	62.96

TAB. 2.5 – Les solutions entières trouvées par la coopération BP+LNS.

La ligne « Heuristique visites multiples » correspond à l'heuristique gloutonne qui transforme une solution de couverture d'ensembles en une solution de partitionnement d'ensembles que nous avons décrite à la fin de la Section 2.3.1. Cette heuristique est surtout utile au début du processus d'optimisation. En effet, la borne supérieure devient rapidement trop serrée et il n'est alors plus possible de visiter les clients plusieurs fois.

Composant	Série 1		Série 2	
	%s	%c	%s	%c
Problème maître relâché	0.04	11.81	0.09	7.14
PLNE	5.40	7.27	3.75	2.38
Heuristique visites multiples	35.71	6.81	45.83	4.36
Total ILOG DISPATCHER	24.84	74.09	35.05	86.11
... à partir d'une solution trouvée par :				
Branch-and-price	63.73	26.36	50.54	18.25
ILOG DISPATCHER	18.58	47.72	32.38	67.85

TAB. 2.6 – Les solutions entières trouvées par la coopération BP+DISPATCHER.

Les deux algorithmes de recherche locale que nous avons employés utilisent chacun un modèle où chaque client est visité exactement une seule fois. L'heuristique de visites multiples ne peut donc améliorer des solutions trouvées par recherche locale. La ligne « Recherche locale... à partir d'une solution trouvée par *branch-and-price* » se réfère aux résultats de la recherche locale lorsqu'elle est invoquée avec comme solution initiale une solution spontanément entière du problème maître relâché, une solution trouvée par le composant de programmation linéaire en nombres entiers, ou une solution obtenue par l'heuristique de visites multiples en transformant la solution découverte par l'une de ces deux dernières méthodes.

La recherche locale trouve la plus grande partie des solutions sur toutes les séries : l'amélioration de performance constatée pour notre schéma de coopération provient naturellement d'abord des grandes capacités de la recherche locale à trouver des solutions de qualité. Le taux de succès des algorithmes de recherche locale est significativement plus élevé lorsque la solution initiale a été trouvée par *branch-and-price* que lorsque la solution initiale provient de la recherche locale elle-même. Ceci illustre le mécanisme de diversification : lorsque le *branch-and-price* trouve une solution, elle est souvent éloignée du dernier optimum local connu, donc elle a plus de chances de pouvoir être améliorée par recherche locale.

Il est à noter que la qualité des solutions n'est pas prise en compte dans les statistiques des Tableaux 2.5 et 2.6. Une solution trouvée par le composant de PLNE pendant sa limite de temps relativement longue (1 minute) représente généralement une amélioration de la précédente solution plus substantielle que l'amélioration obtenue typiquement pendant un court appel de l'algorithme de recherche locale (avec une limite de temps de 10 secondes). Par exemple, sur les instances de taille 100 de la série RC2, quand le composant de PLNE trouve une nouvelle solution, elle représente une amélioration de 2.30% en moyenne (3.59% en cumulant avec l'amélioration obtenue par l'heuristique de visites multiples), tandis qu'un appel de LNS ou d'ILOG DISPATCHER améliore en cas de succès la solution précédente de seulement 1.48% et 0.98% respectivement. C'est pouquoi la contribution du composant de PLNE à la découverte de solutions entières améliorantes est vraisemblablement sous-évaluée dans les Tableaux 2.5 et 2.6.

2.4.5 Qualité des bornes inférieures

	Série 1		*Série 2*	
Temps	1/2 heure	1 heure	1/2 heure	1 heure
BP 1	-0.80%	-0.76%	-0.37%	-0.34%
BP 2	-0.80%	-0.77%	-0.42%	-0.37%
BP+LNS	-0.80%	-0.77%	-0.38%	-0.35%
BP+DISPATCHER	-0.78%	-0.75%	-0.41%	-0.35%

TAB. 2.7 – Qualité des bornes inférieures (instances de taille 100).

Dans cette section, nous comparons la capacité de chaque méthode à fournir des bornes inférieures sur la fonction objectif et nous évaluons dans quelle mesure notre schéma de coopération conserve la capacité du *branch-and-price* à produire des bornes inférieures de qualité. Rappelons que les algorithmes de recherche locale ne fournissent ni bornes inférieures ni preuves d'optimalité. Le Tableau 2.7 donne l'erreur moyenne relative entre les bornes inférieures obtenues par chaque algorithme étudié et les valeurs de référence des Tableaux 2.1 et 2.2 sur les instances de taille 100 des séries 1 et 2 de Solomon. Rappelons que, sur la série 1, les valeurs de référence sont soit les valeurs des solutions optimales, soit des bornes inférieures potentiellement sous-optimales, donc les chiffres indiqués dans le Tableau 2.7 pour cette série ne sont pas des majorants de l'écart des bornes inférieures obtenues à l'optimum. Il est également à noter que, sur plusieurs instances (une instance de la série 1, 15 instances de la série 2), aucun des algorithmes étudiés ne fournit une borne inférieure sur la fonction objectif en une demi-heure : cette limite de temps est trop courte pour que la génération de colonnes converge au nœud racine. Ces instances n'ont pas été prises en compte pour le calcul de l'écart moyen relatif dans le Tableau 2.7.

La conclusion principale du Tableau 2.7 est que nos algorithmes de coopération entre *branch-and-price* et recherche locale (LNS ou ILOG DISPATCHER) conservent la capacité du *branch-and-price* à fournir des bornes inférieures de qualité. Le Tableau 2.4 indique également que notre schéma de coopération produit environ le même nombre de preuves d'optimalité que le *branch-and-price* pur.

Le Tableau 2.7 montre que la qualité des bornes inférieures obtenues par *branch-and-price* pur évolue peu lorsque la limite de temps est augmentée d'une demi-heure à une heure. Ce phénomène peut être expliqué par notre choix de ne générer que des chemins élémentaires dans le sous-problème. La borne inférieure résultante est déjà de très bonne qualité au nœud racine [27, 26]. De plus, comme aucune coupe n'est ajoutée, la borne inférieure augmente lentement au fur et à mesure de l'exploration de l'arbre de *branch-and-price*. C'est pourquoi les bornes inférieures obtenues en une heure par nos algorithmes hybrides ne peuvent pas être significativement meilleures que les bornes inférieures obtenues en une demi-heure par *branch-and-price* pur. Si l'on ne considère que la qualité des bornes inférieures, c'est une raison pour préférer un mécanisme de coopération plus simple qui partage le temps également entre le *branch-and-price* et la recherche locale. Cependant, pour des modèles de *branch-and-price* différents où la

qualité de la borne inférieure est médiocre au nœud racine, puis augmente rapidement au fur et à mesure que des coupes sont ajoutées ou que l'exploration de l'arbre de *branch-and-price* progresse, notre schéma de coopération produira probablement des bornes inférieures meilleures qu'un simple mécanisme de partage de temps.

2.4.6 Preuve d'optimalité pour deux instances ouvertes

Nous présentons enfin les résultats sur deux instances précédemment ouvertes que nous avons résolues à l'optimalité[1] : R211 et RC204, chacune de taille 50. Le Tableau 2.8 donne pour chaque instance le coût de la solution optimale et le nombre correspondant de véhicules. Le Tableau 2.9 donne le temps en secondes nécessaire pour atteindre la solution optimale (T_{opt}), et prouver également l'optimalité (T_{total}), et le nombre de nœuds explorés dans l'arbre de *branch-and-price*. Nous présentons des résultats pour nos deux algorithmes de *branch-and-price* pur « BP 1 » et « BP 2 », et pour nos deux algorithmes hybrides combinant *branch-and-price* et recherche locale « BP+LNS » et « BP+DISPATCHER ».

Instance	R211.50	RC204.50
Coût optimal	535.5	444.2
Nombre de véhicules	3	3

TAB. 2.8 – Valeurs optimales pour les deux instances que nous avons fermées.

	R211.50			*RC204.50*		
Algorithme	T_{opt}	T_{total}	n	T_{opt}	T_{total}	n
BP 1	115 100	196 868	257	-	-	-
BP 2	103 600	126 648	85	-	-	-
BP+LNS	214 100	300 184	281	152 100	-	-
BP+DISPATCHER	25 900	94 411	85	50 200	84 059	1

TAB. 2.9 – Preuve d'optimalité pour les deux instances que nous avons fermées.

Il est à noter que seul BP+DISPATCHER résout l'instance RC204.50 à l'optimalité. BP+LNS atteint la solution optimale mais n'est pas capable de prouver l'optimalité au terme du temps alloué (une semaine). Le *branch-and-price* pur échoue également à résoudre RC204.50 avec la même limite de temps. « BP 2 » atteint seulement une solution de coût 460.6 (478 pour « BP 1 ») et ne termine pas la génération de colonnes au nœud racine (meilleure relaxation continue : 460.6 pour « BP 2 » ; 478 pour « BP 1 »). Sur ce problème avec fenêtres de temps très étendues, il s'avère extrêmement coûteux de générer des plus courts chemins contraints élémentaires intéressants. Au contraire, ILOG DISPATCHER réussit à trouver des colonnes quasiment optimales en un temps raisonnable et le *branch-and-price* peut ensuite trouver la solution optimale et prouver l'optimalité assez rapidement. Ceci illustre un des points essentiels de notre schéma de coopération : la diversification est utile pour générer des solutions entières mais aussi pour générer des colonnes individuelles de bonne qualité.

[1] L'instance R211.50 a depuis été résolue à l'optimalité également par Irnich et Villeneuve [88].

BP+DISPATCHER résout RC211.50 à l'optimalité plus rapidement que le *branch-and-price* pur. L'accélération est particulièrement visible si l'on compare le temps nécessaire pour atteindre la solution optimale, mais le temps total incluant le temps nécessaire à la preuve d'optimalité est également réduit. Notons cependant que, sur cette instance, BP+LNS ralentit la résolution et explore plus de nœuds que le *branch-and-price* pur « BP 2 », parce qu'il ne trouve pas la solution optimale rapidement et est donc incapable d'élaguer des branches en haut de l'arbre de *branch-and-price*. Le même phénomène est observé pour « BP 1 ».

2.5 Conclusion

Nous avons introduit dans ce chapitre une nouvelle stratégie hybride pour combiner les techniques de recherche locale et le *branch-and-price*. Nous avons montré avec de nombreuses expériences sur le problème de routage de véhicules avec fenêtres de temps que notre schéma de coopération améliore uniformément la capacité du *branch-and-price* à générer rapidement des solutions entières de qualité tout en conservant la capacité du *branch-and-price* à produire de bonnes bornes inférieures sur la fonction objectif. L'amélioration des solutions entières obtenues est en général corrélée avec l'efficacité de l'algorithme de recherche locale mis en œuvre dans la coopération, mais des améliorations significatives peuvent être obtenues même avec un algorithme simple de recherche locale. Il reste à prouver que notre schéma de coopération pourra être appliqué avec autant de succès à des problèmes différents et plus complexes. Nos résultats sur un problème difficile et avec deux algorithmes de recherche locale d'efficacité différente sont néanmoins encourageants.

Notre schéma de coopération généralise trois accélérations existantes du *branch-and-price*. Il peut être vu comme la généralisation au *branch-and-price* des heuristiques pour le *branch-and-cut* car la recherche locale est utilisée ici non seulement pour trouver des solutions entières, mais aussi pour introduire de nouvelles colonnes. Notre schéma de coopération peut être appliqué à n'importe quel modèle de *branch-and-price*. Cependant, il n'est pas générique : un nouvel algorithme de recherche locale spécifique au problème étudié doit être mis en œuvre chaque fois qu'un nouveau modèle de *branch-and-price* est à résoudre. Notre schéma de coopération constitue néanmoins un premier pas vers la transposition au *branch-and-price* des stratégies mises en œuvre pour le *branch-and-cut*. Il est à noter que la grande majorité des stratégies de branchement pour le *branch-and-price* sont également spécifiques au modèle à traiter afin d'être faciles à exprimer pour le problème maître et faciles à intégrer dans l'algorithme de résolution du sous-problème. L'ajout d'une heuristique spécifique ne diminue donc pas sensiblement le niveau de généricité du *branch-and-price*. Dans le prochain chapitre, nous étudierons des stratégies pour la programmation linéaire en nombres entiers qui sont génériques dans la mesure où elles peuvent être appliquées à tout modèle de PLNE sans aucune modification et sans information extérieure.

Du point de vue du développement des algorithmes hybrides, notre schéma de coopération obéit au paradigme de la recherche multiple : les gains d'efficacité et de robustesse sont apportés par la diversification induite par l'utilisation d'algorithmes différents pour résoudre le même problème. Notre schéma de coopération souffre des défauts habituels de cette approche : la complexité logicielle est accrue, le nombre de paramètres à calibrer (notamment, quand et avec quelle limite de temps appeler chaque composant) augmente, un coût fixe lié à l'utilisation de chaque composant est encouru (les composants de recherche locale et de PLNE sont appelés même lorsque l'optimum a déjà été atteint, ce qui ralentit la preuve d'optimalité). Comme nous l'avons annoncé à la fin du premier chapitre dans nos objectifs, nous introduirons dans le prochain chapitre un nouveau paradigme de développement pour les algorithmes hybrides qui contourne la plupart de ces difficultés.

Notons enfin en ce qui concerne le problème de routage de véhicules avec fenêtres de temps que nous obtenons de très bons résultats sur la série 2 des instances de Solomon, mais que nos résultats sur la série 1 ne sont pas aussi compétitifs. Ajouter des coupes comme dans les approches de Cook et Rich [35], Irnich [87] ou Irnich et Villeneuve [88] nous permettrait d'améliorer la qualité des bornes inférieures, donc d'augmenter le nombre d'instances résolues à l'optimalité, en particulier sur la série 1 de Solomon.

Chapitre 3

Une nouvelle heuristique pour des problèmes linéaires quelconques en nombres entiers

Ce chapitre présente la nouvelle heuristique pour des problèmes linéaires quelconques en nombres entiers que nous avons appelée *Relaxation Induced Neighborhood Search* (RINS). Il reprend en grande partie l'article [43]. Ce travail a été réalisé en collaboration avec Edward Rothberg (ILOG CPLEX) et Claude Le Pape. En particulier, la stratégie de *guided dives* et les variantes de l'algorithme de *local branching* sont deux idées d'Edward Rothberg.

3.1 Introduction

Si la programmation linéaire en nombres entiers a fait de nombreux progrès pendant les quinze dernières années, en particulier si les outils commerciaux pour résoudre des problèmes linéaires en nombres entiers sont devenus plus efficaces et plus robustes, il est encore très difficile de résoudre optimalement certains problèmes. Dans de tels cas, et à moins qu'une autre technique exacte telle que la programmation par contraintes s'applique bien au problème, l'utilisateur doit se contenter d'une solution réalisable de bonne qualité. Cependant, même la simple obtention d'une bonne solution réalisable peut s'avérer difficile sur certains modèles, en particulier si le temps de résolution est très limité. Face à ces difficultés, il est intéressant de considérer d'autres techniques telles que les algorithmes de recherche locale qui sont connus pour produire rapidement de très bonnes solutions pour de nombreux problèmes, comme nous l'avons indiqué dans le Chapitre 1. Il semble donc prometteur d'appliquer les concepts de recherche locale (voisinage, intensification, diversification) à la programmation linéaire en nombres entiers pour résoudre des modèles très difficiles.

Pour intégrer la recherche locale à la PLNE, il faut répondre à trois questions. La question la plus importante, qui est aussi la plus difficile, est celle de la définition du voisinage d'une solution donnée. Les voisinages sont presque toujours construits grâce à la structure de haut niveau du problème donné à résoudre. Par exemple, dans le chapitre précédent, le voisinage est construit en changeant la position de certaines visites dans le plan de route des véhicules. Pour utiliser une construction similaire dans le cas d'un programme linéaire quelconque en nombres entiers, il faudrait extraire cette structure de la matrice des contraintes, ce qui constitue un défi majeur. La construction d'un voisinage générique est clairement un problème difficile, mais les bénéfices à en attendre sont néanmoins également clairs : cette construction pourrait alors être mise en œuvre sur n'importe quel modèle de PLNE, sans nécessiter d'autre information que le modèle lui-même. Une question auxiliaire est celle de la taille du voisinage. Comme nous l'avons exposé dans la Section 1.2.6 du Chapitre 1, les approches de définition et d'exploration de grands voisinages sont en général efficaces et robustes, mais définissent le plus souvent leurs voisinages en fonction de la structure du problème à résoudre.

La deuxième question est celle de l'intensification, c'est-à-dire de l'exploration du voisinage. Selon sa taille, le voisinage peut être exploré par énumération complète, de manière heuristique, ou comme il est fréquent dans les approches de *large neighborhood search*, par une recherche arborescente tronquée. Pour la programmation linéaire en nombres entiers, cette troisième possibilité revient à définir le voisinage par un modèle plus contraint et explorer ce *sous-modèle* par *branch-and-cut* standard.

Enfin, la troisième question est celle de la diversification. Pour être efficace, un algorithme de recherche locale doit changer régulièrement le sous-espace de recherche sur lequel il se focalise. En PLNE, un moyen immédiat de diversifier la recherche est d'exploiter les différentes solutions entières engendrées par le processus de *branch-and-cut* pour le modèle d'origine. Cependant, nous nous intéressons ici aux modèles pour lesquels les implémentations existantes de *branch-and-cut* ne réussissent pas à trouver de nombreuses solutions, donc cette approche risque d'être insuffisante.

L'heuristique récente du *local branching* [57] propose une réponse à chacune de ces trois questions. Nous introduisons ici deux nouvelles approches, RINS et *guided dives*, qui répondent chacune d'une manière différente à ces trois questions. Nous les détaillons dans la Section 3.2 où nous discutons également des modifications que nous avons apportées à l'algorithme du *local branching*. Nous décrivons ensuite dans la Section 3.3 les modèles que nous avons utilisés pour comparer ces méthodes. Dans la Section 3.4, nous présentons et nous analysons les résultats numériques de nos expériences qui comparent ces méthodes dans trois contextes différents. Nous montrons dans cette section que les deux stratégies que nous avons introduites sont plus efficaces et plus robustes que les algorithmes préalablement existants. Enfin, dans la Section 3.5, nous résumons les conclusions de notre étude pour en tirer des enseignements valables pour le développement des algorithmes hybrides en général.

Comme dans les chapitres précédents, nous supposons dans toute la suite que nous avons à résoudre un problème de minimisation.

3.2 Méthodes

Nous décrivons ici de manière détaillée les méthodes que nous avons mises en œuvre. Comme nous l'avons noté dans l'introduction, elles sont toutes fondées sur l'idée qu'un voisinage de la solution entière courante a des chances de contenir des solutions entières améliorantes.

3.2.1 *Relaxation induced neighborhood search*

Algorithme

Quand un problème est résolu par *branch-and-cut*, typiquement deux solutions sont disponibles. La solution entière courante, *i.e.*, la meilleure solution entière connue à cet instant, est réalisable pour les contraintes d'intégralité mais n'est pas optimale tant que l'optimum n'a pas été atteint. Inversement, la solution de la relaxation continue au nœud courant est le plus souvent fractionnaire, mais sa valeur pour la fonction objectif est meilleure que celle de la solution entière courante. La solution entière courante et la solution de la relaxation atteignent donc respectivement un seul des deux objectifs suivants : l'intégralité et l'optimisation de la fonction objectif. Il existe clairement des variables qui prennent des valeurs différentes dans ces deux solutions, mais il est important de noter qu'un certain nombre de variables y prennent au contraire les mêmes valeurs. RINS repose sur l'intuition que l'affectation de ces variables forme une solution partielle qui a des chances d'être étendue à une solution complète qui est à la fois entière et de bonne qualité. C'est pourquoi cette heuristique se concentre sur les variables dont la valeur diffère dans la solution entière courante et dans la solution de la relaxation continue : ce sont en effet celles qui paraissent mériter le plus d'attention.

L'algorithme de RINS est donc simple. À un nœud de l'arbre global de *branch-and-cut*, les étapes suivantes sont effectuées :

1. Les variables qui prennent la même valeur dans la solution entière courante et dans la solution de la relaxation continue sont fixées ;
2. Une borne égale à la valeur de la solution entière courante est imposée sur la fonction objectif ;
3. Un sous-modèle de PLNE est résolu sur les variables restantes.

Il est à noter que le modèle d'origine est amélioré au fur et à mesure de la résolution par *branch-and-cut* : des inégalités valides sont ajoutées au modèle et de nouvelles bornes globales sur les variables sont découvertes. Notre sous-modèle exploite ces informations additionnelles. Par contre, nous ne restreignons pas la recherche au sous-arbre courant du modèle global : les bornes ajoutées par la stratégie de branchement dans l'arbre global ne sont pas prises en compte dans le sous-modèle.

Le sous-modèle défini par RINS est potentiellement de grande taille et difficile à résoudre, donc son exploration doit souvent être tronquée, ce qui est mis en œuvre par une limite *nl* sur le nombre de nœuds explorés. Toute solution trouvée pendant la résolution du sous-modèle est par construction également une solution du problème initial. Donc, lorsque l'exploration du sous-modèle est interrompue (parce qu'une preuve d'optimalité ou d'irréalisabilité a été obtenue, ou parce que le nombre maximum de *nl* nœuds a été exploré), la solution entière courante du modèle global est mise à jour avec la meilleure solution entière trouvée dans le sous-modèle (le cas échéant), et l'exploration du modèle global reprend. Il est important de préciser que l'exploration du sous-modèle défini par RINS a pour seul effet de fournir de nouvelles solutions entières.

Les capacités de diversification de RINS sont obtenues automatiquement par les changements de la relaxation continue d'un nœud à l'autre dans l'arbre de *branch-and-cut* global. RINS pourrait être appelé à chaque nœud de l'arbre, mais les voisinages engendrés par les relaxations de nœuds consécutifs sont généralement assez semblables. Il s'est donc avéré préférable d'appliquer RINS uniquement tous les f nœuds, pour un certain nombre $f \gg 1$. Il est à noter que, même si le nombre f choisi est grand, plusieurs nœuds de l'arbre global peuvent engendrer les mêmes sous-modèles pour RINS. De manière similaire, un sous-modèle défini par RINS peut être identique à un sous-arbre de l'arbre de recherche du problème d'origine. Rien n'est prévu dans notre algorithme pour éviter ces duplications.

La force de RINS est d'explorer un sous-espace qui est à la fois voisinage de la solution entière courante et voisinage de la solution de la relaxation continue. Sans information extérieure telle qu'une borne inférieure ou supérieure calculée par un autre moyen, l'algorithme de *branch-and-cut* ne sait pas quelle est, parmi la solution entière courante et la solution de la relaxation continue, la solution la plus proche de la solution entière optimale. Dans notre heuristique, ces deux solutions jouent un rôle parfaitement symétrique, donc si l'une est médiocre, l'autre contribuera automatiquement à créer un voisinage prometteur, et vice-versa. C'est pourquoi RINS peut d'une part améliorer significativement des solutions entières de mauvaise qualité parce qu'il est guidé par la relaxation continue. D'autre part, RINS a des chances d'améliorer la robustesse vis-à-vis d'une relaxation continue trop lâche parce qu'il est également guidé par la solution entière courante.

Il est possible que la solution entière courante ait été trouvée à un nœud de l'arbre très éloigné du nœud courant. Dans ce cas, il est à craindre que le nombre de variables fixées par RINS soit faible, donc que le sous-modèle construit par RINS ne soit pas significativement plus facile à résoudre que le modèle d'origine. Cependant, comme le montre la Figure 3.1, RINS fixe au moins les variables binaires de branchement qui correspondent au plus petit sous-arbre contenant le nœud où a été trouvée la solution entière courante[1] et le nœud de la relaxation courante. De plus, la fixation de

[1] On suppose ici pour simplifier que les éventuelles heuristiques mises en œuvre pour trouver des solutions

certaines variables permet de contraindre le sous-problème sur les variables libres restantes (sans néanmoins permettre de fixer des variables supplémentaires par un simple raisonnement linéaire). Enfin, il est important de noter que la programmation linéaire en nombres entiers est un problème combinatoire, donc même une faible diminution du nombre de variables peut faciliter significativement la résolution du problème. Ce phénomène est connu sous le nom d'implosion combinatoire. Nous donnerons quelques statistiques sur la taille et la difficulté des sous-modèles créés par RINS à la fin de la Section 3.4.3.

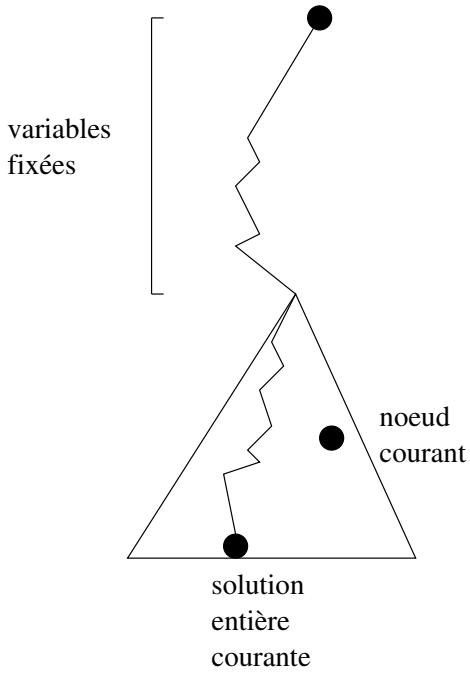

FIG. 3.1 – Un sous-ensemble des variables fixées par RINS.

Relation avec les algorithmes de la littérature

Comme nous l'avons exposé dans la Section 1.3.2 du Chapitre 1, de nombreuses heuristiques pour la PLNE essaient de transformer la solution de la relaxation continue en une solution entière réalisable par arrondis successifs. En d'autres termes, ces heuristiques explorent le voisinage de la solution de la relaxation continue. D'autres heuristiques tentent d'améliorer une solution (réalisable ou non). Mais à notre connaissance, aucune heuristique antérieure n'utilise à la fois la solution de la relaxation continue et la solution entière courante. L'heuristique de programmation linéaire en nombres entiers la plus proche de RINS est sans doute le système TRIP pour les problèmes de couver-

entières respectent les décisions de branchement du nœud où elles sont invoquées.

ture de vols (*crew pairing*) d'American Airlines [6]. Ce système améliore itérativement une solution de couverture d'ensembles (*set partitioning*) en choisissant aléatoirement un ensemble de variables binaires qui prennent la valeur 1 dans la solution entière courante, en fixant temporairement ces variables à 1, puis en reformulant et en résolvant un sous-modèle linéaire en nombres entiers sur les variables restantes (typiquement en agrandissant l'ensemble des variables restantes par génération de colonnes avant de résoudre le sous-modèle). Dans un contexte de couverture d'ensembles, fixer une variable à 1 entraîne la fixation d'un grand nombre d'autres variables à 0 (toutes les variables dont les colonnes correspondantes dans la matrice ont un coefficient non nul sur la même ligne que la variable fixée). Le sous-modèle ainsi construit est donc généralement beaucoup plus facile à résoudre que le problème initial. L'algorithme mis en œuvre dans TRIP est général, mais, à notre connaissance, il n'a été appliqué qu'au problème de couverture d'ensembles.

Si l'on replace maintenant RINS dans le contexte plus général des méthodes existantes de recherche locale, cette heuristique appartient à la classe des méthodes de définition et d'exploration de grands voisinages que nous avons décrite dans la Section 1.2.6 du Chapitre 1. RINS utilise la relaxation continue pour choisir les variables à fixer dans la solution entière courante et n'a besoin d'aucune information extérieure sur la structure du problème. Cette stratégie est donc différente des trois stratégies de définition de grands voisinages préalablement existantes (choix aléatoire, exploitation de la structure du problème, utilisation de l'historique des solutions réalisables) que nous avons mises en évidence page 32. RINS est applicable sans modification à n'importe quel modèle de programmation linéaire en nombres entiers. C'est donc, à notre connaissance, la première méthode de *large neighborhood search* qui soit générique.

RINS est également à rapprocher de la métaheuristique de *path relinking* [71] car il relie dans une certaine mesure deux solutions trouvées au préalable dans des parties différentes de l'arbre global de *branch-and-cut* en explorant l'espace de recherche situé « entre » ces deux solutions. Cependant, deux aspects différencient RINS et le *path relinking*. Premièrement, au lieu de suivre un petit nombre de chemins (souvent un seul) qui mènent d'une solution à l'autre, RINS explore le sous-espace défini par l'intersection de ces deux solutions. Cette exploration est effectuée par un algorithme puissant et complet qui est tronqué afin de contrôler son temps d'exécution. Deuxièmement, et ce point constitue la différence la plus importante avec le *path relinking*, RINS combine les solutions de deux problèmes différents (mais apparentés) : le problème linéaire initial en nombres entiers et sa relaxation continue. Si l'on utilise la terminologie du *path relinking*, une des propriétés essentielles de RINS est que son ensemble (*pool*) de solutions « d'élite » comprend des solutions irréalisables. À notre sens, cette extension permet de s'attaquer à une difficulté inhérente à la programmation linéaire en nombres entiers, l'écart entre la relaxation continue et toute solution entière.

3.2.2 *Local branching*

Le *local branching* (LB) est une stratégie proposée récemment par Fischetti et Lodi [57] pour définir et explorer de manière originale le voisinage d'une solution entière d'un programme linéaire en nombres entiers. Notre heuristique RINS reprend certaines idées introduites dans le cadre du *local branching*, notamment la définition du voisinage comme un sous-modèle de PLNE et son exploration par *branch-and-cut* standard.

Un des points importants du *local branching* est l'utilisation d'une extension de la distance de Hamming. Étant donné deux vecteurs x et x^*, dont les composantes indicées par un certain sous-ensemble B ne peuvent prendre que les valeurs 0 ou 1, la semi-distance utilisée par le *local branching* est :

$$H(x,x^*) = \sum_{j \in B} |x_j - x^*_j|$$

Le *local branching* construit un sous-modèle du modèle d'origine qui ne considère qu'un voisinage autour de la solution entière courante x^* en introduisant la contrainte additionnelle (linéarisée) $H(x,x^*) \leq r$, pour un certain paramètre r appelé *rayon du voisinage*. Si ce voisinage contient effectivement des solutions améliorantes, alors elles seront potentiellement trouvées pendant la résolution de ce sous-modèle par *branch-and-cut*. Comme pour RINS, le sous-modèle construit par *local branching* comprend toutes les coupes et les nouvelles bornes globales sur les variables qui ont été découvertes pendant l'exploration de l'arbre global de *branch-and-cut*. De même, il ne prend pas en compte les bornes locales sur les variables qui sont introduites par la stratégie de branchement.

Il est à noter que seules les composantes binaires interviennent dans la semi-distance H. La contrainte de *local branching* peut donc être linéarisée sous la forme :

$$\sum_{j \in B \cap \{j | x^*_j = 1\}} (1 - x_j) + \sum_{j \in B \cap \{j | x^*_j = 0\}} x_j \leq r$$

La semi-distance H pourrait être étendue de manière naturelle aux variables entières générales, mais la linéarisation de la contrainte de *local branching* résultante nécessiterait l'introduction de nombreuses variables entières. Les contraintes de *local branching* utilisées dans l'article initial [57] et dans nos expériences ne prennent en compte que les variables binaires.

Précisons maintenant quelques détails d'implémentation. Le sous-modèle défini par *local branching* risque d'être encore difficile à résoudre optimalement. Comme pour RINS, nous imposons donc une limite *nl* sur le nombre de nœuds explorés dans chaque sous-modèle. Cependant, contrairement à ce qui se passe en cas d'échec de RINS, nous utilisons la même stratégie que Fischetti et Lodi si le sous-modèle atteint le nombre maximal de nœuds sans avoir généré une nouvelle solution entière. Plus précisément, nous divisons le rayon du voisinage par 2 et résolvons le nouveau sous-modèle de PLNE correspondant. Cette réduction de la taille du voisinage continue jusqu'à ce

que le rayon soit inférieur à 5, ou que l'exploration du sous-modèle produise une nouvelle solution entière. Lorsque le sous-modèle produit une solution, son exploration est continuée jusqu'au terme de la limite des *nl* nœuds, puis le rayon du voisinage est réinitialisé à sa valeur par défaut et le processus est recommencé à partir de la nouvelle solution. Ainsi, plusieurs sous-modèles successifs peuvent être construits et explorés. L'exploration de l'arbre global reprend quand un sous-modèle de rayon $r \leq 5$ ne produit pas de solution améliorante avant la limite des *nl* nœuds. Nous utilisons des critères d'arrêt différents pour RINS et pour le *local branching* car il est important pour le *local branching* d'exploiter le plus possible toute possibilité d'améliorer la solution entière courante. En effet, contrairement à RINS, le *local branching* ne peut être appelé que lorsqu'une nouvelle solution entière est trouvée dans l'arbre de recherche global.

Un point supplémentaire à préciser concerne les solutions découvertes dans l'arbre de recherche global à partir desquelles le *local branching* est appelé. Nous appelons le *local branching* à chaque nœud où une nouvelle solution entière est trouvée et nous construisons le voisinage autour de la meilleure solution entière trouvée à ce nœud. Les heuristiques présentes dans ILOG CPLEX trouvent souvent plusieurs solutions successives à un même nœud (en particulier au nœud racine), mais nous avons conclu expérimentalement qu'il est plus efficace d'explorer uniquement le voisinage de la meilleure de ces solutions.

La première différence majeure entre notre implémentation et celle de Fischetti et Lodi est l'organisation de l'arbre de recherche global. Nous utilisons le *local branching* uniquement comme une heuristique permettant de trouver de meilleures solutions entières. Fischetti et Lodi l'utilisent également comme métastratégie de branchement. Plus précisément, à chaque sous-modèle défini par *local branching* avec la contrainte $H(x,x^*) \leq r$ correspond un sous-modèle complémentaire défini par la contrainte inverse $H(x,x^*) \geq r+1$. Si le premier sous-modèle est exploré complètement (*i.e.*, si une preuve d'optimalité ou d'irréalisabilité est obtenue), alors la recherche subséquente peut être restreinte au sous-modèle inverse. Cette approche a pour avantage principal de ne pas dupliquer l'exploration du voisinage de x^*. Son principal inconvénient réside dans l'augmentation du coût de résolution de chaque nœud au fur et à mesure que les contraintes de *local branching* inverses s'accumulent. Cette dégradation de performance est particulièrement à surveiller parce que les contraintes de *local branching* sont denses. Dans nos expériences numériques, l'avantage procuré par l'absence de redondance dans l'exploration s'est révélé insuffisant pour compenser la réduction du nombre de nœuds pouvant être explorés en un temps donné.

La deuxième différence majeure entre nos implémentations respectives concerne la stratégie de diversification. Dans notre implémentation, la diversification est apportée uniquement par les solutions entières successives trouvées pendant l'exploration de l'arbre de *branch-and-cut* global. Fischetti et Lodi n'exploitent pas ces solutions mais engendrent à la place des solutions entières moins bonnes que la solution entière courante et explorent les voisinages de *local branching* associés. Nos résultats expéri-

mentaux suggèrent que l'utilisation des solutions trouvées dans l'arbre global est plus efficace. Nous montrerons dans la Section 3.4.5 que RINS et *guided dives* exhibent des capacités de diversification encore supérieures.

La question évidente est maintenant de savoir si nous avons conservé les propriétés d'origine du *local branching* en dépit de nos modifications. Nous avons comparé les performances de notre implémentation avec celles de l'implémentation de Fischetti et Lodi, chacune construite à partir de la même version du logiciel ILOG CPLEX (8.1), et notre implémentation s'est révélée uniformément plus efficace. Sur nos modèles de test, l'écart entre la borne inférieure et la meilleure solution entière obtenue était en moyenne de 20% pour notre implémentation, et de 30% pour celle de Fischetti et Lodi. Les résultats de notre implémentation surpassent également ceux de l'implémentation de Fischetti et Lodi à partir d'une version précédente de CPLEX (7.0) qui avaient été présentés dans l'article introduisant le *local branching* [57].

La deuxième question naturelle que le lecteur est en droit de se poser est pour quelles raisons nous avons modifié l'algorithme du *local branching*, alors que nous aurions pu nous contenter de comparer nos résultats avec ceux publiés par Fischetti et Lodi. RINS et la stratégie *guided dives* décrite dans la section suivante utilisent un arbre de *branch-and-cut* standard, il nous a donc semblé naturel de les comparer avec une implémentation du *local branching* qui utilise également un arbre de *branch-and-cut* standard. En particulier, intégrer RINS et le *local branching* dans le même arbre de recherche permet de nous concentrer exclusivement sur la comparaison entre la fixation forte (*hard fixing* mise en œuvre par RINS) et la fixation faible (*soft fixing* mise en œuvre par le *local branching*) en termes de création de sous-modèles intéressants.

Tous les résultats rapportés dans les sections suivantes ont été obtenus avec notre implémentation du *local branching*. Dans toute la suite, sauf mention explicite du contraire, le terme *local branching* désigne l'implémentation particulière que nous venons de décrire.

3.2.3 *Guided dives*

Algorithme

Contrairement aux autres méthodes étudiées dans ce chapitre, la deuxième stratégie que nous introduisons ici ne considère pas des voisinages explicitement et ne construit pas des sous-modèles de PLNE à résoudre récursivement par *branch-and-cut*. Cette stratégie procède de manière alternative en modifiant légèrement la manière dont l'arbre de recherche est exploré.

À chaque nœud d'un arbre de *branch-and-cut*, deux décisions essentielles doivent être prises pour atteindre *in fine* une feuille de l'arbre. La première concerne le choix de la variable de branchement. La deuxième consiste à choisir le nœud à explorer immédiatement parmi les nœuds fils créés lors du branchement. Dans notre stratégie *guided dives* (littéralement, plongeons dirigés), nous fondons ce deuxième choix sur la valeur

de la variable de branchement dans la solution entière courante. Nous choisissons le nœud fils dans lequel la variable de branchement binaire est fixée à la valeur qu'elle prend dans la solution entière courante. (Dans le cas d'une variable de branchement entière générale, nous choisissons le nœud fils dans lequel la variable de branchement peut encore prendre la valeur qu'elle prend dans la solution entière courante). *Guided dives* revient à explorer implicitement un voisinage de la solution entière courante.

Cette stratégie peut être mise en œuvre très simplement, il suffit d'interroger à chaque nœud la valeur de la variable de branchement dans la solution entière courante. Pourtant, la simplicité de cet algorithme ne l'empêche pas d'être très efficace, comme nous le montrerons à la Section 3.4.

Relation avec les algorithmes de la littérature

À notre connaissance, l'idée d'utiliser une solution réalisable connue pour modifier l'exploration de l'arbre de recherche n'a pas été souvent employée en programmation linéaire en nombres entiers ou en programmation par contraintes. Nous pouvons citer des méthodes de programmation par contraintes mises en œuvre pour résoudre des problèmes d'ordonnancement qui utilisent la solution courante pour guider la recherche [98], ou comme préférence dans les problèmes de réordonnancement pour calculer une nouvelle solution avec un nombre minimal de changements [89, 145].

3.3 Modèles étudiés

Nous avons rassemblé 37 modèles pour lesquels il est difficile de trouver de bonnes solutions entières. Toutes les expériences décrites dans les sections suivantes ont été effectuées sur ce jeu de tests. Ces modèles proviennent de quatre sources :

- Cinq problèmes d'ordonnancement d'atelier avec coûts d'avance et retard. Ces problèmes `ljb` ont été largement utilisés dans la littérature des algorithmes génétiques (GA), comme le rapporte par exemple la comparaison de différents algorithmes présentée par Vázquez et Whitley [153]. Nous utilisons la modélisation disjonctive, déjà employée notamment par Applegate et Cook [7]. Ce problème (modélisation et résolution) sera étudié plus en détail dans le Chapitre 4.
- Onze problèmes de synthèse de réseaux et de routage de flots extraits d'un plus grand jeu de données construit à partir de données industrielles qui a été introduit par Chabrier *et al.* [28]. Sur ces modèles `rococo`, le *branch-and-cut* donne des résultats médiocres comparée à d'autres méthodes telles que le *branch-and-price* ou une algorithme de grands voisinages à base de programmation par contraintes (CP+LNS) [28].
- Vingt modèles parmi les vingt-quatre modèles étudiés dans l'article introduisant le *local branching* [57] (qui spécifie l'origine de chacun de ces modèles). Nous avons supprimé deux modèles de ce jeu de données : `rail507` et `rail2586c` car la dernière version de CPLEX les résout facilement à l'optimalité, ce qui leur

enlève tout intérêt pour notre étude. Nous avons également retiré le modèle `van` pour lequel la plus grande difficulté est d'ordre numérique et non combinatoire. Nous avons enfin enlevé le modèle `NSR8K` car aucun algorithme que nous avons testé, y compris l'implémentation d'origine du *local branching* par Fischetti et Lodi, ne trouve une solution entière pour ce problème, même en un temps très long (cinq heures) sur notre ordinateur de référence (les résultats initiaux de Fischetti et Lodi ont été en fait obtenus en désactivant le *presolve* de CPLEX).

– Un modèle, `swath`, de la bibliothèque MIPLIB 3.0 [17].

Ces modèles sont disponibles auprès des auteurs de [43], les deux derniers groupes de modèles sont également disponibles sur le web.[2]

Le Tableau 3.1 énumère différentes statistiques pour chaque instance, notamment le nombre total de variables (n), le nombre de variables binaires (b), le nombre de variables entières générales (i), le nombre de contraintes (m), la valeur de l'objectif de la meilleure solution que nous connaissons, l'écart relatif entre cette borne supérieure et la meilleure borne inférieure connue (gap) et l'algorithme qui a généré la meilleure solution.

Pour avoir une meilleure idée de l'amélioration potentiellement atteignable sur ces modèles, nous avons inclus dans ce tableau quelques résultats qui n'ont pas été obtenus au cours des expériences décrites ici. Par exemple, nous avons souvent donné à nos algorithmes une limite de temps beaucoup plus longue que celle que nous spécifierons dans la Section 3.4.2 si nous avions des raisons de penser que de meilleures bornes supérieures ou inférieures pourraient être atteintes grâce à ce temps supplémentaire. Pour les modèles de synthèse de réseaux et d'ordonnancement, les meilleures solutions ont souvent été obtenues par des algorithmes spécifiques. Nous mentionnons dans le Tableau 3.1 les instances où la meilleure solution n'a pas été obtenue pendant les expériences décrites ci-après avec la référence de l'algorithme qui a obtenu cette meilleure solution. LB+RINS 1 et LB+RINS 2 sont des algorithmes combinant RINS et *local branching* que nous décrirons dans la Section 3.4.1. Les bornes inférieures utilisées dans le calcul de la colonne *gap* sont les meilleures bornes obtenues pendant les expériences de la section 3.4 ou avec une limite de temps plus longue comme mentionné précédemment. Les meilleures bornes inférieures pour les modèles de production `tr12-30`, `A1C1S1`, `A2C1S1`, `B1C1S1`, et `B2C1S1` proviennent de la thèse de M. Van Vyve [149].

Il est à noter que les méthodes considérées ici ont été spécifiquement prévues pour des modèles difficiles à résoudre. Toutes dégradent les performances, souvent de manière significative, sur des modèles où il n'est pas difficile de trouver des solutions entières de bonne qualité. La plupart des modèles de PLNE ne sont pas assez difficiles pour mettre en valeur les capacités de ces méthodes à trouver de bonnes solutions entières.

[2] `http://www.or.deis.unibo.it/research_pages/ORinstances/MIPs.html` et `http://www.caam.rice.edu/~bixby/miplib/miplib.html`

Instance	n	b	i	m	Meilleure solution connue	gap	Meilleur algorithme
A1C1S1	3648	192	0	3312	11557.09	20.96%	LB+RINS 1
A2C1S1	3648	192	0	3312	10889.14	13.51%	LB+RINS 2
B1C1S1	3872	288	0	3904	24544.25	20.58%	LB+RINS 1-long
B2C1S1	3872	288	0	3904	25740.15	20.08%	LB+RINS 1-long
arki001	1388	415	123	1048	7580813.0459	0.00%	LB
biella1	7328	6110	0	1203	3065084.57	0.03%	RINS, LB+RINS 2
glass4	322	302	0	396	1460013800.0	40.90%	LB+RINS 1
net12	14115	1603	0	14021	214	61.23%	RINS, guided dives, LB+RINS 1
nsrand_ipx	6621	6620	0	735	51360	1.36%	RINS, guided dives, LB+RINS 1, LB+RINS 2
rail2586c	13226	13215	0	2589	953	1.78%	LB
rail4284c	21714	21705	0	4287	1071	1.57%	LB
rail4872c	24656	24645	0	4875	1550	2.50%	LB+RINS 1
roll3000	1166	246	492	2295	12890	3.39%	LB+RINS 1
seymour	1372	1372	0	4944	423	2.68%	RINS
sp97ar	14101	14101	0	1761	662671913.92	1.15%	LB+RINS 2
sp97ic	12497	12497	0	1033	429562635.68	1.19%	LB+RINS 1
sp98ar	15085	15085	0	1435	529814784.7	0.45%	LB
sp98ic	10894	10894	0	825	449144758.40	0%	LB+RINS 1-long
tr12-30	1080	360	0	750	130596	0.05%	CPLEX par défaut et autres stratégies
UMTS	2947	2802	72	4465	30122200	3.01%	LB+RINS 1-long
swath	6805	6724	0	884	471.03	25.17%	LB-F&L 8.1
rococoB10-011000	4456	4320	136	1667	19449	25.71%	CP+LNS — par déduction [28]
rococoB10-011001	4456	4320	136	1677	21265	32.87%	CP+LNS — par déduction [28]
rococoB11-010000	12376	12210	166	3792	32246	33.23%	CP+LNS — par déduction [28]
rococoB11-110001	12431	12265	166	8148	42444	50.86%	CP+LNS [28]
rococoB12-111111	9109	8778	331	8978	39831	53.84%	LB+RINS 1-long
rococoC10-001000	3117	2993	124	1293	11465	1.51%	RINS, LB+RINS 1
rococoC10-100001	5864	5740	124	7596	16664	35.31%	CP+LNS, CPLEX par défaut — par déduction [28]
rococoC11-011100	6491	6325	166	2367	20889	21.46%	CP+LNS [28]
rococoC11-010100	12321	12155	166	4010	20889	28.80%	CP+LNS — par déduction [28]
rococoC12-111100	8619	8432	187	10842	35909	11.34%	CP+LNS — par déduction [28]
rococoC12-100000	17299	17112	187	21550	35512	14.58%	CP+LNS [28]
ljb2	771	681	0	1482	0.507679	51.09%	RINS, LB+RINS 1, LB+RINS 2
ljb7	4163	3920	0	8133	0.133655	70.63%	RINS
ljb9	4721	4460	0	9231	0.739	84.44%	GA [153]
ljb10	5496	5196	0	10742	0.512	64.70%	GA [153]
ljb12	4913	4633	0	9596	0.399	88.29%	GA [153]

TAB. 3.1 – Les modèles.

3.4 Résultats expérimentaux

3.4.1 Méthodes testées

Nous présentons maintenant les résultats de nos expériences où nous avons testé les méthodes décrites dans la Section 3.2 sur les modèles présentés dans la Section 3.3. Toutes les expériences ont été réalisées à partir d'ILOG CPLEX 8.1 sur un Pentium IV cadencé à 2GHz avec le système d'exploitation Linux. Pour rappel, les méthodes comparées sont les suivantes :

- CPLEX avec la paramétrisation par défaut (le lecteur est invité à se reporter à la synthèse [18] pour plus de détails sur les stratégies de *branch-and-cut* utilisées par défaut dans CPLEX) ;
- *Relaxation induced neighborhood search* (RINS), tel que décrit à la Section 3.2.1 ;
- Notre implémentation de l'algorithme de *local branching* que nous avons détaillée à la Section 3.2.2 ;
- La stratégie *guided dives* que nous avons présentée à la Section 3.2.3.

Comme nous l'avons mentionné précédemment dans les Sections 3.2.1 et 3.2.2 respectivement, RINS et le *local branching* utilisent chacun deux paramètres dont il faut initialiser la valeur. Pour RINS, nous devons choisir la fréquence f à laquelle l'heuristique est appelée, et le nombre nl de nœuds à explorer dans le sous-modèle. Pour le *local branching*, nous devons choisir la valeur initiale r du rayon du voisinage, et le nombre nl de nœuds à explorer dans le sous-modèle. Nous avons conduit un certain nombre d'expériences avec différentes paramétrisations, et il s'est avéré que les résultats variaient très peu pour RINS, et étaient légèrement moins stables pour le *local branching* (voir l'Annexe A pour les détails chiffrés). Pour l'ensemble des expériences rapportées dans la suite, nous avons choisi les paramètres qui semblaient donner les meilleurs résultats : $f = 100$ et $nl = 1000$ pour RINS ; $r = 10$ et $nl = 1000$ pour le *local branching*.

Il est possible de construire de nombreux hybrides en combinant RINS, *local branching* et *guided dives*. La stratégie de *guided dives* pourrait par exemple être appliquée tout simplement au sous-modèle construit par RINS. C'est un sujet de recherche en soi d'explorer exhaustivement toutes les combinaisons possibles. Cependant, pour avoir une idée des améliorations possibles, nous avons testé deux hybrides naïfs qui sont susceptibles de compenser les faiblesses de chacune de ces trois méthodes. Le premier algorithme hybride, que nous appelons LB+RINS 1, active à la fois RINS et *local branching* dans le même arbre de recherche. De cette façon, l'heuristique du *local branching* est utilisée pour améliorer les solutions trouvées par RINS (ainsi que les autres solutions trouvées au cours de l'optimisation du modèle global). Plus précisément, RINS est appelé tous les f nœuds. quand l'exploration du sous-modèle construit par RINS se termine, le *local branching* est appelé sur la dernière solution trouvée par RINS (si une solution améliorante a été trouvée). Le *local branching* est également appelé directement quand une nouvelle solution entière est trouvée dans l'arbre glo-

bal. Dans les deux cas, quand le *local branching* se termine, l'optimisation du modèle global est reprise avec une éventuelle mise à jour de la solution entière courante. Ainsi la prochaine invocation de RINS utilisera peut-être une solution entière découverte par *local branching*. Le but principal de cet hybride est d'augmenter à la fois le nombre et la qualité des voisinages explorés par *local branching*. Notre deuxième algorithme hybride, LB+RINS 2, ajoute une contrainte de *local branching* (avec $r = 20$) à chaque sous-modèle de RINS. Le but est ici de limiter la difficulté du sous-modèle dans les cas où la relaxation et la solution entière courante sont significativement différentes. Pour chacun de ces algorithmes hybrides, les critères d'arrêt de RINS et du *local branching* sont les mêmes qu'en pur RINS ou pur *local branching*. Les résultats de ces deux méthodes seront présentés en même temps que les autres.

3.4.2 Méthodologie

Avant d'analyser les résultats de nos expériences, précisons comment ces résultats vont être présentés. Nous avons choisi d'exprimer la qualité des solutions par le ratio de la valeur de la solution obtenue par une méthode donnée divisée par la valeur de la meilleure solution connue pour ce modèle (donnée dans le Tableau 3.1). Ce ratio est donc toujours plus grand que 1.

Nos expériences ont produit un volume important de données (6 méthodes différentes sur 37 modèles différents, chaque expérience produisant une suite plus ou moins longue de solutions entières). Il était donc nécessaire de résumer ces données pour les présenter de manière lisible et permettre une comparaison des performances de chaque méthode sur l'ensemble de nos modèles de tests. Nous capturons cette performance agrégée par la moyenne géométrique sur tous les modèles du ratio décrit ci-dessus (valeur de la solution obtenue par une méthode donnée / valeur de la meilleure solution connue).

Quel temps de calcul allouer à chaque méthode ? Une limite de temps trop courte pourrait tronquer la suite de solutions entières successives obtenue par l'une des méthodes bien avant qu'elle n'ait eu le temps de converger. Ceci est particulièrement vrai pour le *local branching* pour lequel une solution améliorante produit souvent une suite de sous-modèles dont la résolution n'est pas instantanée. Inversement, une limite de temps trop longue pourrait brouiller les différences de performance entre les différents algorithmes, en particulier si l'un d'entre eux atteint la solution optimale au début du temps alloué. Nous avons trouvé qu'une limite de temps d'une heure établit un équilibre entre ces deux arguments pour la grande majorité de nos modèles de test. En effet, la plupart des méthodes trouvent souvent des solutions améliorantes en-deçà de cette limite, mais ne progressent plus significativement au-delà. Ce phénomène sera visible sur les graphiques présentés à la section suivante.

Nous avons enfin choisi de regrouper nos modèles en trois sous-ensembles afin d'éviter d'agréger les résultats sur des modèles trop disparates. Les modèles de « faible écart » sont ceux pour lesquels l'écart relatif entre la plus mauvaise solution obtenue par

l'une des 6 méthodes étudiées ici et la meilleure solution connue est inférieur à 10%. Les modèles d'« écart intermédiaire » sont ceux pour lesquels cet écart se situe entre 10% et 100%, et les modèles d'« écart important » sont ceux pour lesquels cet écart est supérieur à 100%. Nous avons alloué aux modèles d'« écart important » un temps plus long (deux heures au lieu d'une) parce que les grandes différences de performance sur ces modèles suggèrent que les résultats peuvent être amenés à changer si une limite de temps plus grande est considérée.

3.4.3 Optimisation à partir de zéro

Nous présentons maintenant les résultats obtenus quand les méthodes précédemment décrites optimisent les modèles de notre jeu de tests sans aucune information préalable. Ces expériences sont destinées à évaluer les performances globales de chaque méthode. Nous détaillerons dans les deux prochaines sections des expériences plus ciblées pour évaluer la capacité de chaque méthode à améliorer une solution donnée et à diversifier la recherche.

Le Tableau 3.2 présente la qualité des solutions obtenues par chaque méthode au terme du temps alloué, exprimée par le ratio de la valeur de la solution trouvée par la méthode divisée par la valeur de la meilleure solution connue. Le meilleur ratio obtenu sur chaque modèle (et les éventuels ratios *ex-aequo*) est mis en évidence en gras.

Ce sont les résultats de ce tableau qui ont été utilisés pour classifier les modèles en trois classes selon leur difficulté apparente. Notons que le groupe d'« écart intermédiaire » comprend essentiellement des modèles `rococo`, tandis que la classe d'« écart important » regroupe tous les modèles d'ordonnancement `ljb` sauf un. Il est intéressant de remarquer que la taille des modèles n'est pas le déterminant primordial de l'écart de performance observé. Les modèles du sous-ensemble d'« écart important » sont d'ailleurs parmi les modèles les plus petits de notre jeu de tests.

Les Figures 3.2 à 3.4 montrent comment la qualité des solutions évolue en fonction du temps.

Les conclusions que nous pouvons tirer de ces données expérimentales sont les suivantes. D'abord, RINS est la méthode la plus efficace. Cette heuristique obtient des solutions de qualité avant les autres méthodes, et maintient son avance même lorsque plus de temps est alloué à chaque méthode. Bien que moins efficace que RINS, notre stratégie *guided dives* donne également de bons résultats et ses performances sont significativement supérieures à celles de la stratégie par défaut de CPLEX. Enfin, le *local branching* est en moyenne légèrement meilleur que la stratégie par défaut de CPLEX, mais ses performances se dégradent significativement sur les modèles d'« écart important ».

Nos deux méthodes hybrides LB+RINS 1 et LB+RINS 2 sont plus efficaces que le *local branching* pur, mais moins efficaces que RINS pur. En d'autres termes, il ne semble pas intéressant de combiner les méthodes de cette manière. Cependant, comme nous l'avons déjà mentionné, les trois méthodes que nous considérons ici peuvent être

Instance	CPLEX par défaut	RINS	Local branching	Guided dives	LB+ RINS 1	LB+ RINS 2
Problèmes de « faible écart » — une heure						
A1C1S1	1.039	1.006	1.011	1.016	**1.000**	1.002
A2C1S1	1.038	1.007	**1.000**	1.035	1.006	**1.000**
arki001	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
B2C1S1	1.070	1.041	1.079	1.047	**1.010**	1.024
biella1	1.004	**1.000**	1.001	**1.000**	1.001	**1.000**
nsrand_ipx	1.006	**1.000**	1.003	**1.000**	**1.000**	**1.000**
rail2586c	1.015	1.016	1.014	1.018	**1.007**	1.019
rail4284c	1.007	1.009	1.007	1.005	**1.002**	1.009
rail4872c	1.010	1.008	1.008	1.014	**1.000**	1.006
rococoB10-011000	1.041	1.022	1.034	1.022	**1.008**	1.044
rococoB10-011001	1.041	1.033	**1.025**	1.083	1.101	1.079
rococoB11-010000	**1.042**	1.061	1.118	**1.042**	1.119	1.093
rococoC10-001000	1.001	1.001	1.001	1.001	**1.000**	**1.000**
roll3000	1.014	1.005	1.003	1.016	**1.000**	1.001
seymour	1.012	**1.002**	1.009	1.005	**1.002**	**1.002**
sp97ar	1.012	1.001	1.012	1.006	1.005	**1.000**
sp97ic	1.026	1.002	1.012	1.013	**1.000**	1.006
sp98ar	1.007	**1.002**	**1.002**	1.003	**1.002**	1.003
sp98ic	1.015	1.003	1.004	1.004	**1.002**	1.005
trl2-30	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
UMTS	1.001	**1.000**	1.001	1.001	**1.000**	1.001
Problèmes d'« écart intermédiaire » — une heure						
B1C1S1	1.118	**1.011**	1.020	1.096	1.018	1.013
glass4	1.123	1.096	1.130	1.113	**1.000**	1.119
ljb2	1.225	**1.000**	1.116	1.038	**1.000**	**1.000**
net12	1.192	**1.000**	1.192	**1.000**	**1.000**	**1.000**
rococoB11-110001	1.205	**1.121**	1.254	1.224	1.216	1.148
rococoB12-111111	1.141	**1.029**	1.057	1.099	1.084	1.062
rococoC10-100001	1.180	**1.095**	1.214	1.132	1.137	1.294
rococoC11-011100	1.312	**1.055**	1.330	1.118	**1.055**	1.452
rococoC11-010100	1.173	1.081	1.081	1.123	**1.053**	1.060
rococoC12-100000	1.270	**1.096**	1.110	1.268	1.117	1.278
rococoC12-111100	1.148	**1.025**	1.125	1.112	1.070	1.081
swath	1.222	1.048	1.154	1.093	**1.031**	1.048
Problèmes d'« écart important » — deux heures						
ljb7	2.375	1.061	1.580	1.582	**1.028**	1.255
ljb9	1.858	**1.581**	1.995	1.718	1.646	1.809
ljb10	1.601	**1.212**	1.693	1.295	1.226	1.500
ljb12	2.568	**1.512**	3.083	2.012	2.097	2.418

TAB. 3.2 – Ratio solution obtenue / meilleure solution connue.

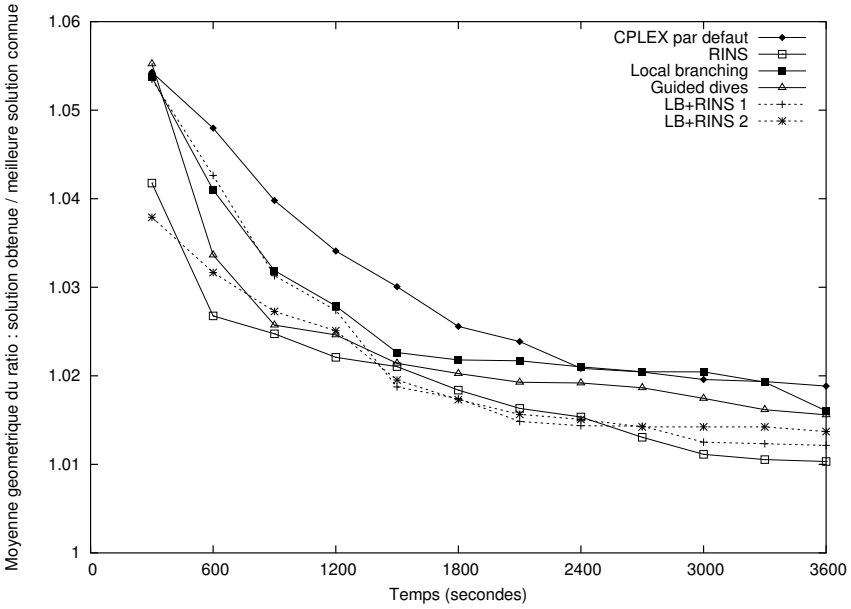

FIG. 3.2 – Problèmes de « faible écart ».

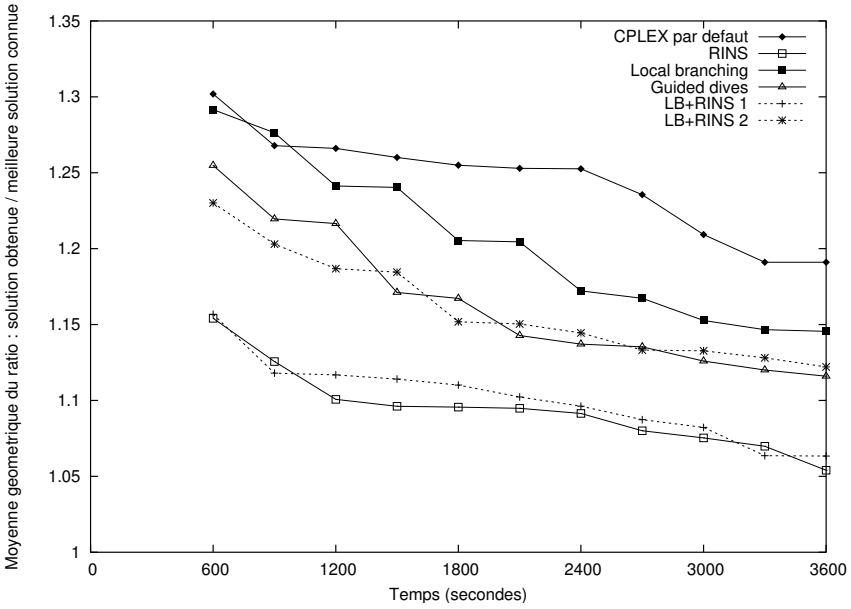

FIG. 3.3 – Problèmes d'« écart intermédiaire ».

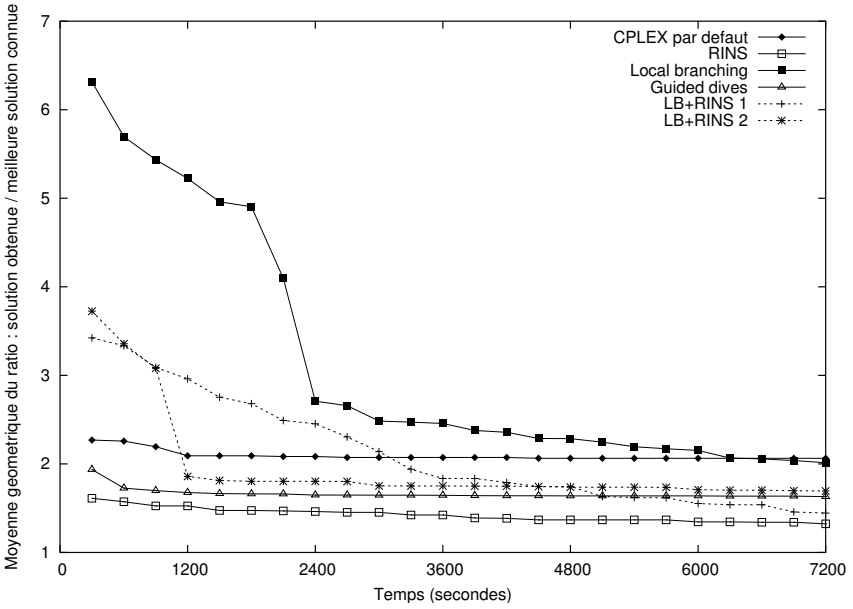

FIG. 3.4 – Problèmes d'« écart important ».

combinées de diverses manières, donc il n'est pas du tout exclu qu'une autre combinaison donne de meilleurs résultats.

Revenons plus en détail sur le comportement du *local branching* sur les modèles d'« écart important ». Ces mauvaises performances peuvent sans doute être expliquées par la qualité médiocre des premières solutions générées. En effet, RINS, le *local branching* et *guided dives* sont tous inactifs tant que le *branch-and-cut* dans l'arbre global n'a pas trouvé une première solution entière, et tous sont invoqués la première fois à partir de cette solution initiale. La première solution générée par ILOG CPLEX pour ces modèles d'« écart important » est en général médiocre car CPLEX a pour but de produire une solution réalisable le plus tôt possible. Les voisinages définis par *local branching* autour de cette solution médiocre ont donc peu de chances de contenir de bonnes solutions. Leur exploration produira donc des solutions de qualité également mauvaise, dont les voisinages seront ensuite explorés. Ce processus conduit à la définition et à l'exploration d'une longue suite de sous-modèles peu prometteurs. RINS n'est pas aussi sensible à la qualité des premières solutions trouvées dans l'arbre global pour deux raisons. Premièrement, la solution entière courante et la solution de la relaxation continue jouent des rôles symétriques pour RINS, donc une solution entière médiocre n'implique pas nécessairement la définition d'un voisinage médiocre. Deuxièmement, le sous-modèle défini par RINS contient moins de variables que le sous-modèle défini par *local branching*, et ne contient pas non plus la contrainte dense ajoutée par *local branching*. Donc, un même nombre de nœuds est généralement exploré plus rapidement dans le sous-modèle de RINS que dans le sous-modèle de *lo-*

cal branching. La Figure 3.4 montre clairement que brancher dans l'arbre global de *branch-and-cut* (comme c'est le cas par défaut dans CPLEX et dans la stratégie *guided dives*) permet d'améliorer rapidement des solutions initiales de mauvaise qualité. Il est à noter que le point le plus à gauche sur la Figure 3.4 représente la meilleure solution connue après 5 minutes, et non la toute première solution trouvée. On peut donc raisonnablement s'interroger sur les possibilités d'amélioration du *local branching* qui seraient obtenues en l'invoquant seulement après la découverte par *branch-and-cut* standard d'une assez bonne solution. Le problème d'une telle approche est de reconnaître à partir de quel seuil une solution est de qualité suffisante. Ce type de décision est particulièrement ardu sur des modèles difficiles à résoudre parce que la borne inférieure fournie par la relaxation continue est souvent éloignée de la valeur optimale de l'objectif, et ne peut donc être utilisée de manière fiable pour évaluer la qualité des solutions entières. Notons par ailleurs sur la Figure 3.4 que la solution trouvée au bout de 5 minutes par la stratégie par défaut de CPLEX est certes bien meilleure que la solution obtenue par *local branching*, mais elle est déjà significativement dépassée par la solution trouvée par RINS pendant le même laps de temps.

Observons maintenant la taille des sous-modèles définis par RINS et leur difficulté de résolution. Le Tableau 3.3 donne pour chaque modèle le nombre moyen de variables des sous-modèles définis par RINS (exprimé comme une fraction de la taille du modèle d'origine), le temps passé dans les sous-modèles (exprimé comme une fraction du temps total alloué pour l'optimisation), et le nombre de sous-modèles définis et explorés par RINS avant la limite de temps. Comme nous l'avons mentionné à la fin de la Section 3.2.1, il est théoriquement possible que les solutions entières et que les relaxations diffèrent sur un nombre significatif de variables, ce qui conduirait RINS à définir des sous-modèles de grande taille et coûteux à résoudre. Mais nous avons constaté expérimentalement que la taille des sous-modèles et le temps passé dans RINS étaient en fait très peu corrélés. Comme souvent en programmation linéaire en nombres entiers, la difficulté d'un sous-modèle ne dépend pas uniquement de sa taille. Plus anecdotiquement, la Figure 3.5 montre l'évolution au cours du temps du nombre de variables du sous-modèle (exprimé comme précédemment comme une fraction de la taille du modèle d'origine) et le nombre de nœuds explorés dans le sous-modèle (exprimé comme une fraction de la limite maximum *nl* imposée sur l'exploration du sous-modèle). Nous avons recueilli les chiffres pour les 400 premiers sous-modèles construits par RINS sur deux instances : `ljb7` et `sp98ic`. Remarquons que le nombre de variables présentes dans le sous-modèle varie peu au cours du temps. Pour `ljb7`, l'exploration du sous-modèle est arrêtée dans la grande majorité des cas parce que la limite sur le nombre de nœuds est atteinte. Par contre, pour `sp98ic`, l'exploration est généralement arrêtée bien avant cette limite car le sous-modèle est résolu à l'optimalité ou, cas beaucoup plus fréquent, parce qu'une preuve d'irréalisabilité est rapidement obtenue. Ces deux exemples montrent que les caractéristiques des sous-modèles construits par RINS varient beaucoup et que nous sommes encore loin de bien les comprendre.

Instance	Taille des sous-modèles	Temps passé dans les sous-modèles	Nombre de sous-modèles
A1C1S1	0.882	0.910	35
A2C1S1	0.891	0.906	35
arki001	0.328	0.660	1260
B2C1S1	0.873	0.827	14
biella1	0.077	0.004	192
nsrand_ipx	0.026	0.306	858
rail2586c	0.126	0.429	9
rail4284c	0.096	0.263	4
rail4872c	0.104	0.312	3
rococoB10-011000	0.109	0.104	331
rococoB10-011001	0.111	0.262	252
rococoB11-010000	0.062	0.123	48
rococoC10-001000	0.095	0.068	308
roll3000	0.423	0.126	1111
seymour	0.398	0.606	65
sp97ar	0.012	0.182	343
sp97ic	0.005	0.107	1380
sp98ar	0.009	0.144	601
sp98ic	0.006	0.151	887
tr12-30	0.590	0.353	2788
UMTS	0.134	0.506	741
B1C1S1	0.880	0.884	23
glass4	0.298	0.495	23856
ljb2	0.199	0.806	1700
net12	0.915	0.001	10
rococoB11-110001	0.057	0.142	42
rococoB12-111111	0.054	0.154	80
rococoC10-100001	0.095	0.068	65
rococoC11-011100	0.109	0.178	56
rococoC11-010100	0.139	0.234	28
rococoC12-100000	0.041	0.037	36
rococoC12-111100	0.043	0.043	144
swath	0.013	0.050	5056
ljb7	0.098	0.739	780
ljb9	0.110	0.794	547
ljb10	0.138	0.825	428
ljb12	0.142	0.839	359
Moyenne géométrique sur			
... toutes les instances	0.109	0.202	-
... groupe de « faible écart »	0.092	0.211	-
... groupe d'« écart intermédiaire »	0.141	0.113	-
... groupe d'« écart important »	0.121	0.798	-

TAB. 3.3 – Statistiques sur les sous-modèles définis par RINS.

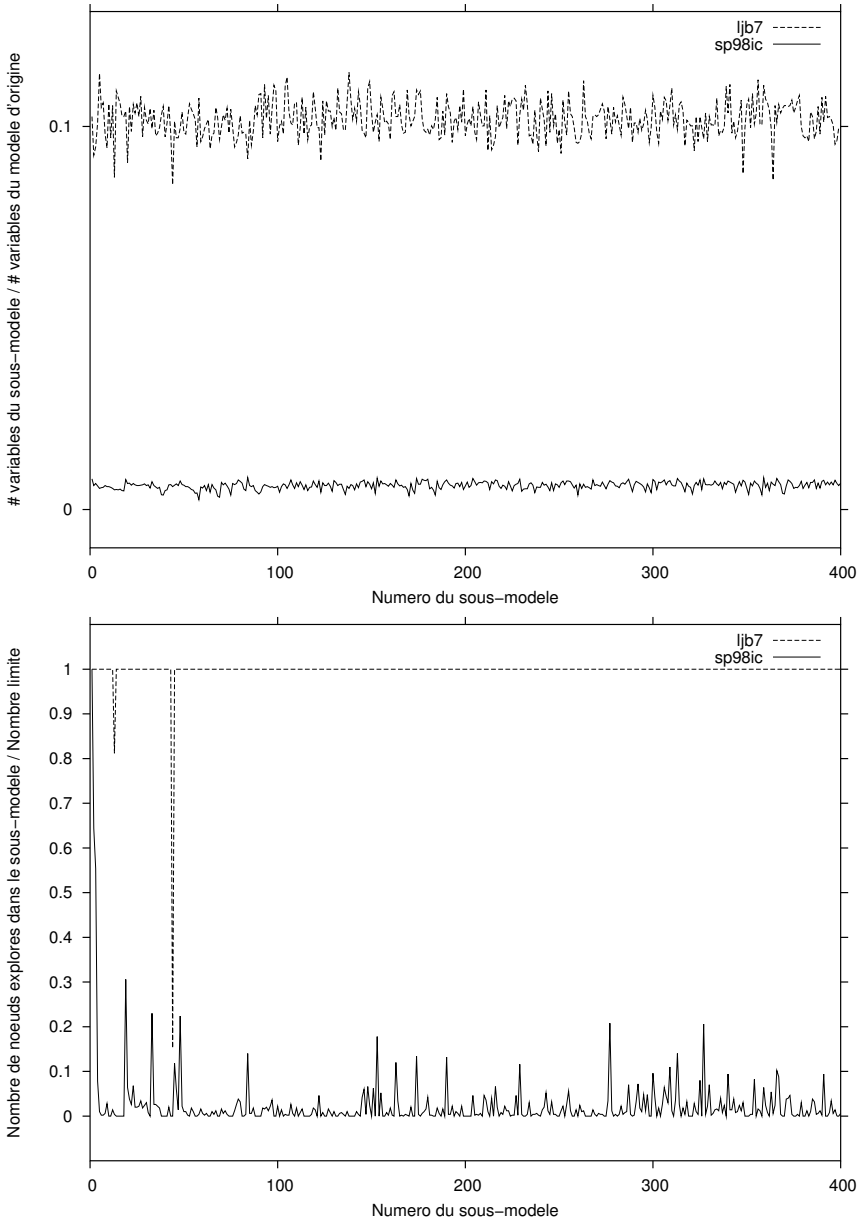

FIG. 3.5 – Évolution du nombre de variables et du nombre de nœuds des sous-modèles définis par RINS pour `ljb7` et `sp98ic`.

Pour finir, quand RINS ne trouve pas certaines solutions particulières d'excellente qualité, nous voudrions comprendre les raisons de cet échec. Deux explications peuvent être avancées : les voisinages définis par RINS ne contiennent pas ces solutions, ou bien les sous-modèles sont trop difficiles à résoudre et l'échec provient de l'inefficacité de l'exploration des voisinages. Nous avons conduit une expérience simple en considérant une solution donnée que RINS n'avait pas trouvée, et en testant si les voisinages définis par RINS contiennent cette solution. Il s'est avéré que c'est rarement le cas. Certes, les voisinages définis par RINS peuvent contenir d'autres solutions améliorantes qui n'ont pas été découvertes par l'exploration tronquée des sous-modèles, mais notre conclusion est que les améliorations futures de RINS sont plus susceptibles de venir du changement de la définition des voisinages plutôt que du changement de la méthode employée pour les explorer.

3.4.4 Amélioration d'une solution donnée

Nous avons conduit des expériences plus détaillées afin de progresser dans notre connaissance des forces et des faiblesses des différentes stratégies présentées, et dans l'espoir de construire de meilleurs algorithmes hybrides grâce à ces informations. Nous avons identifié trois facteurs nécessaires pour obtenir un algorithme globalement efficace. Le premier est la capacité à améliorer rapidement une solution médiocre. Ce facteur a été mis en évidence par les mauvaises performances du *local branching* sur les modèles d'« écart important ». Le deuxième facteur est la capacité à continuer à améliorer une solution de bonne qualité. Le dernier facteur est la capacité à diversifier la recherche pour générer de nouveaux voisinages dans lesquels la recherche sera à nouveau intensifiée. Si les différents algorithmes exhibent des propriétés différentes pour chacun de ces facteurs, alors cette information pourrait être exploitée pour construire une nouvelle stratégie hybride efficace.

Cette section considère la capacité de chaque méthode à améliorer une solution entière donnée. Plus précisément, pour chaque modèle, deux solutions initiales ont été utilisées :

- Une solution de qualité médiocre : l'une des premières solutions entières obtenues avec la stratégie par défaut de CPLEX ;
- Une solution de bonne qualité : la solution obtenue avec la stratégie par défaut de CPLEX en une heure de calcul.

Pour chaque algorithme (y compris la stratégie par défaut de CPLEX), la solution initiale est utilisée comme *MIP start* en haut d'un nouvel arbre de *branch-and-cut*. Ceci ne correspond pas exactement à l'amélioration d'une solution donnée telle que rencontrée pendant nos expériences de la section précédente parce que les informations sur les branches déjà construites et éventuellement élaguées avant de trouver cette solution ne sont pas prises en compte ici. Mais ces informations sont ignorées exactement de la même façon pour chaque algorithme.

Pour chaque modèle, la limite de temps a été fixée à la durée (arrondie au multiple

de 100 secondes immédiatement supérieur) nécessaire pour que le *local branching* finisse d'optimiser la première série de sous-modèles construite à partir de la solution initiale. Comme une itération de *local branching* est typiquement plus longue que l'exploration d'un sous-modèle de RINS, la limite de temps que nous avons choisie permet à chaque méthode de finir au moins une itération. Bien que RINS bénéficie en plus en théorie de l'exploration de l'arbre global quand l'exploration de son premier sous-modèle est terminée, en pratique, sur les instances que nous avons étudiées, le *branch-and-cut* standard sur le modèle d'origine trouve rarement des solutions entières. Donc l'amélioration en termes de solutions entières que nous mesurons ici pour RINS est bien due essentiellement à l'exploration de ses sous-modèles.

	CPLEX par défaut	RINS	Local branching	Guided dives
À partir d'une solution médiocre				
... toutes les instances	39.44%	**44.98%**	35.83%	42.16%
... groupe de « faible écart »	26.96%	**28.03%**	24.56%	26.97%
... groupe d'« écart intermédiaire »	38.91%	**48.39%**	40.03%	44.08%
... groupe d'« écart important »	77.94%	**83.74%**	66.38%	81.19%
À partir d'une bonne solution				
... toutes les instances	0.28%	**3.95%**	1.99%	2.82%
... groupe de « faible écart »	0.00%	**0.64%**	0.62%	0.25%
... groupe d'« écart intermédiaire »	0.18%	2.30%	**2.53%**	1.08%
... groupe d'« écart important »	2.04%	**23.62%**	7.36%	19.69%

TAB. 3.4 – Pourcentage moyen d'amélioration de la qualité des solutions.

Le Tableau 3.4 présente l'amélioration moyenne de la qualité de la solution. Ces chiffres sont obtenus en calculant la moyenne géométrique de l'objectif de la solution obtenue au terme du temps alloué divisé par la valeur de la solution initiale, ôtée de 1. Une amélioration de 50% signifie donc que la valeur de l'objectif a été réduite de moitié. Ces données montrent que RINS est, parmi les méthodes étudiées, la plus efficace pour améliorer une solution médiocre. Ce résultat était attendu puisque l'utilisation de la relaxation continue pour définir les voisinages est à même de compenser la mauvaise qualité des solutions entières. Notre tentative d'explication des mauvaises performances du *local branching* sur les instances `ljb` présentée à la section précédente est confirmée ici : si l'on considère la capacité à améliorer une solution de mauvaise qualité, le *local branching* est significativement moins efficace que RINS et que *guided dives*, et parfois seulement légèrement meilleur que la stratégie par défaut de CPLEX. De manière plus surprenante, quand il s'agit d'améliorer une solution de bonne qualité, le Tableau 3.4 montre également que le *local branching* n'est jamais significativement supérieur à RINS ou à *guided dives*, et même parfois très inférieur.

3.4.5 Diversification de la recherche

Dans cette section, nous considérons un autre facteur déterminant pour l'efficacité de chaque algorithme : le temps nécessaire pour produire une solution améliorante ;

en d'autres termes, la puissance de diversification de chaque méthode. Rappelons que notre implémentation du *local branching* repose pour la diversification sur l'exploration de l'arbre global par *branch-and-cut*, donc ses capacités de diversification sont identiques à celles du *branch-and-cut* standard (ici la stratégie par défaut de CPLEX). Nous ne présentons donc pas de chiffres pour le *local branching* dans cette section.

	CPLEX par défaut	RINS	Guided dives
À partir d'une solution médiocre			
Nombre d'échecs (parmi 37) : aucune solution trouvée en 1/2 heure	1	**0**	1
Temps médian (en secondes) pour obtenir la première solution améliorante			
... dans les 36 cas où tous les algorithmes réussissent	36.48	**24.98**	30.59
À partir d'une bonne solution			
Nombre d'échecs (parmi 37) : aucune solution trouvée en 1/2 heure	32	**4**	14
Temps médian (en secondes) pour obtenir la première solution améliorante			
... dans les 5 cas où tous les algorithmes réussissent	284.44	**9.09**	53.19
... dans les 23 cas où seuls RINS et *guided dives* réussissent	-	**99.56**	119.74

TAB. 3.5 – L'efficacité des schémas de diversification.

Le Tableau 3.5 présente les capacités de diversification de chaque algorithme, exprimées par la médiane sur un sous-ensemble de modèles du temps nécessaire à l'obtention d'une solution améliorante. Nous avons utilisé comme solution initiale les mêmes solutions, respectivement de mauvaise et de bonne qualité, que dans la section précédente et nous avons fixé une limite de temps d'une demi-heure pour chaque méthode et chaque modèle. Ce tableau montre que RINS est la méthode la plus robuste, avec le plus petit nombre d'échecs. C'est aussi la méthode la plus rapide, que la solution initiale soit de bonne ou de mauvaise qualité.

Les expériences relatées dans les deux dernières sections nous ont certes permis de montrer que RINS est la méthode la plus efficace, quelle que soit la capacité spécifiquement mesurée. Mais, ces résultats offrent malheureusement peu d'informations pour combiner les différentes méthodes de manière plus satisfaisante que nos hybrides LB+RINS 1 et LB+RINS 2.

3.5 Conclusion

3.5.1 Récapitulation des contributions à la programmation linéaire en nombres entiers

Nous avons introduit dans ce chapitre deux nouvelles stratégies qui trouvent de bonnes solutions entières pour des modèles difficiles de programmation linéaire en nombres entiers : une heuristique que nous avons appelée *relaxation induced neighborhood search* (RINS) et une stratégie d'exploration arborescente que nous avons appelée *guided dives*. Ces deux algorithmes sont génériques, ils peuvent être appliqués à n'importe quel modèle de PLNE, sans nécessiter d'autre information que le modèle

lui-même.

RINS améliore l'heuristique existante de *local branching* sur plusieurs aspects :

– Le *local branching* définit un voisinage autour de la solution entière courante uniquement. Au contraire, les voisinages définis par RINS exploitent de manière symétrique et simultanée la solution entière courante et la solution de la relaxation continue, ce qui permet à RINS d'améliorer rapidement des solutions entières médiocres et d'être robuste vis-à-vis de relaxations de mauvaise qualité.
– Le *local branching* ne peut être invoqué que lorsqu'une nouvelle solution entière est trouvée. Au contraire, RINS peut être invoqué à chaque nœud de l'arbre global de *branch-and-cut*, car la diversification est apportée par les changements de la relaxation continue de nœud en nœud. Cette diversification automatique est intéressante en particulier sur les modèles où un *branch-and-cut* standard ne trouve que rarement de nouvelles solutions entières.
– Un sous-modèle défini par RINS est exploré plus rapidement qu'un sous-modèle défini par *local branching* : il contient moins de variables et ne contient pas la contrainte dense ajoutée par *local branching*.

Nous avons montré avec des expériences numériques complètes et détaillées sur de nombreux modèles difficiles que RINS est plus efficace que le *local branching*, *guided dives* et la stratégie par défaut de CPLEX, à la fois pour produire de bonnes solutions entières à partir de zéro, pour améliorer une solution donnée de bonne ou de mauvaise qualité, et pour diversifier la recherche. *Guided dives* est la deuxième stratégie la plus efficace pour ces trois critères et améliore également significativement les résultats de la stratégie par défaut de CPLEX.

RINS et *guided dives* sont implémentés dans ILOG CPLEX 9.0. RINS est invoqué avec le paramètre `IloCplex::MIPEmphasis=4`, et *guided dives* est invoqué avec le paramètre `IloCplex::DiveType=3`.

3.5.2 Extensions de RINS

L'algorithme de RINS est actuellement très peu sophistiqué. Nous avons considéré plusieurs modifications pour tenter d'améliorer ses performances.

Selon les résultats de l'expérience que nous avons rapportée à la fin de la Section 3.4.3, la voie qui semble la plus prometteuse est de changer la définition du voisinage pour l'agrandir, c'est-à-dire de ne pas fixer dans le sous-modèle un certain nombre de variables qui prennent la même valeur dans la relaxation au nœud courant et dans la solution entière courante. Une manière simple est de choisir ces variables aléatoirement. Ce processus permettrait également d'obtenir facilement une diversification supplémentaire et de pallier le déterminisme total de l'implémentation actuelle de RINS. Une autre manière possible est de ne fixer que les variables qui prennent la même valeur dans la solution entière courante et dans les relaxations continues des n précédents nœuds. Nous avons implémenté ces deux modifications sans obtenir d'amélioration uniforme.

Le deuxième type d'amélioration possible, qui nous semble moins prometteur, concerne la résolution des sous-modèles. Nous avons essayé de les résoudre avec le paramètre de CPLEX qui donne la priorité à la découverte de solutions entières plutôt qu'à la preuve de l'optimalité, `IloCplex::MIPEmphasis=1`, sans obtenir non plus de changement notable dans les résultats.

Le troisième type d'amélioration possible, qui nous semble plus hypothétique, consiste à partager plus d'information entre la résolution du modèle global et la résolution des différents sous-modèles. En particulier, il pourrait être intéressant de réutiliser dans les sous-modèles les informations obtenues pendant la résolution du modèle global, par exemple la structure de l'arbre de *branch-and-cut*, la frontière des nœuds ouverts, les branches déjà élaguées, ... Par ailleurs, il s'avère qu'au fur et à mesure des progrès de la recherche dans l'arbre global, les sous-modèles définis par RINS sont souvent irréalisables. Il serait intéressant d'utiliser ces informations d'une manière ou d'une autre dans l'arbre global, par exemple en différant l'exploration des sous-arbres contenant les nœuds correspondants, ou en ajoutant des coupes pour exclure ces affectations partielles. Cependant, nos expériences avec les coupes inverses de *local branching* ont montré que partager plus d'information entre modèle global et sous-modèle n'accélérait pas nécessairement la résolution.

Enfin, la limite principale de RINS est que cet algorithme nécessite une première solution entière. Or, RINS fonctionne lorsqu'une solution entière médiocre est disponible, donc il devrait être possible de changer la définition du voisinage (prendre en compte les variables qui prennent la même valeur dans plusieurs relaxations continues par exemple ?) pour transformer RINS en une heuristique de première solution.

3.5.3 Les algorithmes *conceptuellement hybrides*

Si l'on se place maintenant dans le cadre plus général des algorithmes hybrides, le travail effectué sur les heuristiques pour la programmation linéaire en nombres entiers nous a permis de formaliser une nouvelle classe d'algorithmes hybrides. Nous appelons algorithme *conceptuellement hybride* un algorithme qui ne met en œuvre qu'une seule technique de résolution mais exploite et transpose dans ce cadre les concepts de plusieurs techniques. En d'autres termes, les algorithmes conceptuellement hybrides réalisent une intégration conceptuelle plutôt qu'une coopération logicielle. RINS, *guided dives* et *local branching* sont des algorithmes conceptuellement hybrides : la seule technique qu'ils mettent en œuvre est le *branch-and-cut* et ils transposent les concepts de la recherche locale (intensification, diversification, et avant tout le concept de voisinage) pour les intégrer à cette technique de résolution.

Les algorithmes conceptuellement hybrides constituent un paradigme de développement qui contourne la plupart des difficultés rencontrées lors du développement des algorithmes hybrides décrits dans la littérature que nous avons détaillées à la Section 1.1.3 du Chapitre 1. En effet, comme une seule technique de résolution est utilisée, les coûts fixes associés à la mise en œuvre de plusieurs composantes et les coûts

de communication entre les différentes composantes ne sont plus encourus. Deuxièmement, la complexité logicielle des algorithmes conceptuellement hybrides est comparable à celle des algorithmes purs. Notons par exemple qu'aucun paramètre n'est nécessaire pour *guided dives*, et comme le montre l'Annexe A, il est facile de calibrer les deux paramètres de RINS pour obtenir des résultats de bonne qualité quel que soit le type de problème considéré. Troisièmement, comme le montre la généricité des trois algorithmes que nous avons étudiés dans ce chapitre, les algorithmes conceptuellement hybrides s'appliquent plus facilement à une large classe de problèmes que les algorithmes hybrides existants qui sont souvent spécialisés pour un problème très particulier. Enfin, les algorithmes hybrides bénéficient facilement des progrès réalisés pour la technique d'optimisation sous-jacente.

Algorithme	Techniques	Concepts	Conceptuellement hybride ?
Décomposition	– Technique 1 : problème maître – Technique 2 : sous-problème	Concepts des techniques 1 et 2	Non
Recherche multiple	– Technique 1 : composant 1 – Technique 2 : composant 2	Concepts des techniques 1 et 2	Non
Large Neighborhood Search	– Recherche locale (?) : définition du voisinage – Recherche arborescente : exploration du voisinage	– Recherche locale : voisinage, intensification, diversification – Recherche arborescente : décision, propagation	Oui / Non
RINS	*Branch-and-cut*	– Recherche locale : voisinage, intensification, diversification – PLNE : relaxation continue – Exploration arborescente : fixation forte	Oui
Guided dives	*Branch-and-cut*	– Recherche locale : voisinage – Exploration arborescente : plonger vers des nœuds prometteurs	Oui
Local branching (implémentation initiale de M. Fischetti et A. Lodi)	*Branch-and-cut*	– Recherche locale : voisinage, intensification, diversification – PLNE : inégalités valides (fixation faible) – Exploration arborescente : diviser pour régner	Oui

TAB. 3.6 – Quels algorithmes hybrides sont conceptuellement hybrides ?

Pour illustrer cette idée, le Tableau 3.6 passe en revue quelques algorithmes hybrides et indique leur appartenance ou non à cette nouvelle classe d'algorithmes. Nous avons déjà mentionné que RINS, *guided dives*, et le *local branching* sont des algorithmes conceptuellement hybrides. À l'opposé, les schémas de décomposition et de

recherche multiple que nous avons mis en évidence dans le Chapitre 1 ne sont pas des algorithmes conceptuellement hybrides. Leur fonctionnement repose sur la mise en œuvre de plusieurs composants logiciels, chacun résolvant un problème donné par une technique différente, et sur l'échange d'information entre ces composants.

La classification des approches à base de grands voisinages (*large neighborhood search*) est moins immédiate. Une technique de résolution (exploration arborescente : PLNE, PPC) est employée pour explorer le voisinage. Mais, on pourrait ajouter qu'une technique différente (recherche locale) est utilisée pour construire le voisinage. En effet, reprenons les trois manières de définir un grand voisinage que nous avons formalisées au Chapitre 1, page 32. Si le grand voisinage est défini en fixant des variables choisies au hasard, on peut difficilement alléguer que ce choix relève d'une technique de recherche locale, même si les choix aléatoires sont souvent employés en recherche locale à des fins de diversification. Par contre, si le voisinage est défini en exploitant la structure de haut niveau du problème ou l'historique des solutions réalisables découvertes préalablement, alors cette technique relève typiquement de la recherche locale, même si la structure du problème est parfois également exploitée dans les stratégies de branchement, notamment en programmation par contraintes ou en *branch-and-price*. Par ailleurs, si le voisinage est construit non en fixant certaines variables explicites, mais en ajoutant des contraintes supplémentaires (ce qui revient à fixer des variables implicites), comme c'est souvent le cas en ordonnancement, alors cette technique est plus apparentée à l'ajout de contraintes redondantes en programmation par contraintes ou à l'ajout d'inégalités valides en programmation linéaire en nombres entiers qu'aux techniques de recherche locale. Enfin, si le voisinage est de type k-OPT (avec k potentiellement grand), alors la construction est équivalente à l'ajout d'une contrainte de type *local branching* qui impose que le nombre de variables qui peuvent prendre une valeur différente de leur valeur dans la solution courante est inférieur à k. Selon la définition du grand voisinage et la manière dont il est effectivement construit, l'appartenance formelle des approches à base de grands voisinages à la classe des algorithmes conceptuellement hybrides peut donc être discutée. Il est cependant indéniable que les algorithmes de *large neighborhood search* en sont très proches.

Nous espérons que d'autres algorithmes seront développés selon le paradigme des algorithmes conceptuellement hybrides. Un sujet de recherche qui semble particulièrement prometteur est la transposition à la programmation linéaire en nombres entiers du concept de conflit issu de la programmation par contraintes, comme l'illustrent les travaux récents de Demassey [48] autour de *resolution search* [31], ainsi que les travaux en cours de Codato et Fischetti sur les coupes combinatoires de Benders [33] et les travaux de Chu et Xia sur les coupes entières de Benders [30].

Chapitre 4

Application au problème d'ordonnancement d'atelier avec coûts d'avance et de retard

Ce chapitre revient plus en détail sur le problème d'ordonnancement d'atelier avec coûts d'avance et retard pour lequel nous avons déjà donné quelques résultats dans le chapitre précédent. Nous nous sommes concentrés sur ce problème parce que c'est un problème difficile et peu étudié, pour lequel il n'existe notamment pas de bon modèle de programmation linéaire en nombres entiers et sur lequel les deux stratégies introduites au chapitre précédent sont particulièrement efficaces. Ce travail a été réalisé dans l'équipe « Manufacturing Scheduling » d'ILOG, en particulier avec Claude Le Pape, et a été présenté à l'occasion de plusieurs conférences [44, 42, 39].

4.1 Introduction

4.1.1 La problématique du « juste-à-temps »

En ordonnancement, l'objectif le plus classique est de minimiser la date de fin de la dernière opération, autrement dit de minimiser la durée totale de l'ordonnancement (*makespan*). Récemment, une attention croissante a été accordée à la minimisation de coûts d'avance et retard. À chaque opération est associée une date échue (*due date*) et si l'opération se termine effectivement avant ou après cette date, un coût est encouru. Cette nouvelle fonction objectif modélise en particulier la problématique de « juste-à-temps », par exemple pour la gestion des stocks. En effet, un retard dans l'accomplissement d'une opération a des conséquences négatives sur la satisfaction du client et la date de mise sur le marché du produit. Inversement, si une opération est terminée plus tôt que prévu, alors il faut stocker le produit avant de passer à l'étape

suivante, ce qui pose notamment le problème des produits périssables qui ne peuvent pas être stockés plus de t périodes de temps, des coûts de stockage, de l'organisation du transport du lieu de fabrication au lieu de stockage, et de la capacité maximale des entrepôts ou plus généralement de leur aménagement. L'objectif « juste-à-temps » peut être modélisé de plusieurs manières, par exemple en prenant en compte les coûts de stockage et la durée totale de l'ordonnancement [60, 127]. Si les coûts de stockage ne sont comptabilisés que sur le produit fini, alors il suffit de pénaliser la différence (positive ou négative) entre la date de fin de la dernière opération de chaque tâche et sa date échue. C'est le problème que nous considérons ici.

Pénaliser le fait que des opérations terminent en avance est parfois controversé car la valeur d'une solution peut alors être améliorée en insérant des temps morts sur certaines machines, ce qui est en contradiction avec l'utilisation maximale des capacités de production qui est un critère intuitif de la qualité d'un ordonnancement. Cependant, les problèmes de production sont souvent complexes et deux problèmes d'horizons différents sont à distinguer. Le premier problème est de décider à long terme quelles capacités de production mettre en place pour assurer un certain débit de production. Le deuxième problème est de décider au jour le jour l'ordonnancement de chaque opération (à capacités de production fixées) avec pour objectif de minimiser les coûts d'avance et de retard. De plus, comme nous l'exposerons plus précisément dans le paragraphe suivant, des poids différents peuvent être associés aux coûts d'avance et de retard, ce qui permet par exemple d'attribuer plus d'importance au critère de retard. Enfin, la stratégie de résolution peut se décomposer en deux phases pour émuler une optimisation multicritères où il existe une hiérarchie entre le critère d'avance et le critère de retard, la première phase minimisant les coûts de retard, la deuxième phase minimisant les coûts d'avance tout en n'autorisant qu'une dégradation limitée des coûts de retard par rapport à la première phase.

4.1.2 Définition du problème

Nous considérons le problème d'ordonnancement d'atelier $n \times m$ défini sur un ensemble J de n tâches et un ensemble R de m machines de capacité unitaire. Chaque tâche $j \in J$ se compose d'un ensemble de m opérations non-préemptives ordonnées suivant une permutation donnée $\sigma_j = (\sigma_{j1}, \sigma_{j2}, \ldots, \sigma_{jm})$ des machines : la tâche j doit d'abord être exécutée sur la machine σ_{j1}, puis sur la machine σ_{j2}, et ainsi de suite. Notons p_{ji} la durée d'exécution de la tâche j sur la machine i. La tâche j est également caractérisée par sa date de disponibilité r_j avant laquelle elle ne peut commencer, sa date échue d_j (on suppose dans toute la suite sans perte de généralité que $0 \leq r_j \leq d_j$) et deux coefficients positifs de coût, le coefficient d'avance α_j et le coefficient de retard β_j. Notons C_j la date de fin de la dernière opération de la tâche j. Le coût encouru pour

la tâche j est :

$$\begin{cases} \alpha_j(d_j - C_j) & \text{si } C_j \leq d_j \\ \beta_j(C_j - d_j) & \text{si } C_j > d_j \end{cases}$$

L'objectif est de minimiser la somme des coûts encourus pour chaque tâche.

Nous nous plaçons ici dans le cadre de ce modèle très pur pour la clarté des formules qui vont suivre. Mais nous sommes en fait capables de traiter des problèmes où le nombre d'opérations varie suivant les tâches, où les relations de précédence entre deux opérations successives d'une même tâche peuvent s'appliquer indifféremment aux dates de début et aux dates de fin des opérations et peuvent être de plus caractérisées par un délai minimum ou un délai maximum. Nous traitons les contraintes de précédence classiques (chaque opération a au plus un prédécesseur et un successeur) et les contraintes d'assemblage (une opération peut avoir plusieurs prédécesseurs dans le graphe de précédence). Il serait également possible avec un peu plus de travail de traiter des graphes de précédence quelconques. Par contre, ce modèle ne peut s'appliquer par exemple lorsqu'on introduit des coûts de préparation (*setups* dus à des changements d'outils, nettoyage, ...) qui ne vérifient pas l'inégalité triangulaire.

4.1.3 État de l'art

Le problème d'ordonnancement d'atelier avec coûts d'avance et retard est particulièrement difficile parce qu'il contient à la fois des contraintes disjonctives (une opération doit être ordonnancée avant *ou* après une autre opération pour respecter la contrainte de ressources) que la programmation linéaire en nombres entiers ne sait pas traiter efficacement, et un objectif de type somme (somme des coûts d'avance et retard pour toutes les tâches du problème) que la programmation par contraintes ne sait pas propager efficacement. De plus, un changement local (par exemple l'inversion de l'ordre de deux opérations) peut provoquer l'insertion ou la suppression de périodes d'inactivité sur certaines machines, et modifier ainsi profondément la solution globale. Il est donc difficile d'évaluer l'impact d'un changement local sur la fonction objectif. C'est pourquoi ce problème est également difficile à résoudre par recherche locale.

Modèles de programmation linéaire en nombres entiers

Il existe trois grandes manières de modéliser les problèmes d'ordonnancement en programmation linéaire en nombres entiers. Nous ne les présentons ici que de façon superficielle et le lecteur est invité à se référer à la synthèse de Queyranne et Schulz [119] pour une étude polyédrale détaillée de chacun de ces modèles. Ces modèles ont été le plus souvent utilisés pour la minimisation de la durée totale de l'ordonnancement. S'il est généralement possible de modifier simplement la fonction objectif pour minimiser à la place les coûts d'avance et retard, les propriétés de ces modèles, la qualité de la relaxation linéaire par exemple, peuvent s'en trouver modifiées. En outre, il est com-

plexe d'adapter les stratégies mises en œuvre pour la minimisation de la durée totale de l'ordonnancement pour obtenir de bons résultats sur cet objectif modifié.

Le premier modèle est le modèle disjonctif utilisé notamment par Applegate et Cook [7]. À chaque couple d'opérations à exécuter sur la même machine est associé une variable binaire qui correspond à l'ordre d'exécution de ces deux opérations. Ces variables de décisions sont reliées par des contraintes de type *big-M* aux variables de date de début des opérations sur lesquelles sont exprimées les contraintes de précédence. Cette formulation souffre de deux inconvénients majeurs. Premièrement, la présence de contraintes de type *big-M* produit une relaxation continue de mauvaise qualité [53] et rend difficile l'obtention de bonnes solutions entières dans le cas de la minimisation des coûts d'avance et retard [44]. Deuxièmement, le nombre de variables binaires croît avec le carré du nombre d'opérations en disjonction, ce qui est particulièrement problématique si le nombre d'opérations est important, ou si le nombre de machines est réduit (à nombre d'opérations constant).

Le deuxième modèle est le modèle indicé par le temps, appelé aussi modèle en temps discrétisé [119]. À chaque opération et à chaque instant t est associée une variable binaire qui indique si cette opération commence à la date t. La relaxation linéaire de ce modèle est de meilleure qualité que celle du modèle disjonctif [53]. Cependant, le nombre de variables est très important, en particulier dans le cas de la minimisation des coûts d'avance et retard où il est plus difficile de borner l'horizon que dans le cas de la minimisation de la durée totale de l'ordonnancement. Il est cependant possible de contourner partiellement ce problème en n'explicitant pas toutes les variables grâce à la génération de colonnes [144].

Dans le troisième modèle, les variables de décision sont les variables positionnelles. Considérons pour la simplicité de l'exposé le problème à une machine. À chaque opération i et à chaque position p dans la séquence des opérations est associée une variable binaire x_{ip} indiquant si cette opération est la $p^{ème}$ opération à être exécutée. Les variables auxiliaires correspondent à la date de l'événement p, c'est-à-dire la date de début de la $p^{ème}$ opération à être exécutée. Ce type de modèle a d'abord été introduit pour le problème à une machine [97], puis étendu au problème de *permutation flowshop* [119], puis au problème de *flowshop* quelconque [45]. Ce type de modèle comprend moins de variables que le modèle en temps discrétisé et ne contient pas de contraintes de type *big-M*, donc a des chances d'exhiber une relaxation linéaire de meilleure qualité que le modèle disjonctif. Cependant, Dauzère-Pérès et Lasserre [45] notent qu'il semble difficile de généraliser leur modèle à l'ordonnancement d'atelier quelconque sans introduire un nombre important de variables et de contraintes supplémentaires. En effet, dans le cas où l'ordre des machines n'est pas le même pour chaque tâche, il est plus complexe de relier les dates des événements sur des machines différentes pour exprimer les contraintes de précédence entre opérations d'une même tâche. Par ailleurs, contrairement à la minimisation du *makespan*, la minimisation des coûts d'avance et retard n'est pas très efficace dans ce type de modèle. En effet, le lien est

faible entre la date de fin de l'opération en position p et la date échue de cette opération $\sum_i d_i x_{ip}$, expression qui peut prendre un grand éventail de valeurs lorsque le vecteur x est fractionnaire, en particulier s'il existe de grandes disparités parmi les dates échues d_i.

Nous présenterons dans la Section 4.2 les améliorations et les extensions que nous avons apportées au modèle disjonctif. Nous avons choisi ce modèle d'abord parce qu'il permet immédiatement de modéliser le problème d'ordonnancement d'atelier quelconque contrairement au modèle avec variables positionnelles. Ensuite, le modèle en temps discrétisé requiert un nombre de variables beaucoup trop important pour les instances issues de problèmes industriels que nous avons considérées dans nos expériences numériques. Enfin, notre but est de comparer les résultats obtenus avec l'état de l'art et d'évaluer dans quelle mesure les techniques de résolution présentées au chapitre précédent peuvent compenser une modélisation simple telle que la modélisation disjonctive.

Autres approches

Il existe de nombreuses approches pour résoudre le problème à une machine. Deux problèmes différents sont distingués dans la littérature : le placement dans le temps des opérations, leur séquence étant fixée, qui est un problème polynomial ; le choix de la séquence des opérations, qui est un problème NP-difficile. Le placement dans le temps à séquence fixée peut naturellement être obtenu en résolvant un programme linéaire, mais il existe des algorithmes spécifiques pour le problème à une machine [65, 137]. Le choix de la séquence peut être effectué par une combinaison de programmation dynamique et recherche locale [78, 79] ou par *branch-and-bound* [84, 138]. Le lecteur est invité à se référer à [83] pour une synthèse des approches existantes.

Le problème d'ordonnancement d'atelier a été beaucoup moins étudié. À notre connaissance, seul un petit nombre d'algorithmes (heuristiques ou complets) ont été décrits dans la littérature pour traiter ce problème (choix de la séquence et placement dans le temps).

Une première heuristique (HLS : *Hybrid Local Search*) est un algorithme hybride entre recherche locale et programmation linéaire [14]. Un algorithme de recherche tabou détermine la séquence des opérations sur chaque machine ; la programmation linéaire (modèle disjonctif) détermine les dates de début de chaque opération qui sont optimales pour la séquence choisie (c'est-à-dire les variables de séquencement ayant été fixées).

Une deuxième heuristique est un algorithme de définition et d'exploration de grands voisinages à base de programmation par contraintes [42]. Cette heuristique est utilisée pour déterminer la séquence des opérations sur chaque machine et, comme dans HLS, le placement dans le temps de chaque opération est effectué par programmation linéaire. L'une des raisons qui expliquent la bonne performance de cette heuristique est l'adéquation particulière entre la définition de l'un des voisinages et la fonction ob-

jectif de minimisation des coûts d'avance et retard. Pour deux tâches p et q choisies aléatoirement parmi les tâches qui ne finissent pas à l'heure (*i.e.*, exactement à leur date échue), ce voisinage libère toutes les opérations des tâches p et q et, pour chacune de ces opérations, libère également quelques opérations ordonnancées autour de celle-ci sur la même machine, plutôt avant ou après cette opération suivant que la tâche correspondante est respectivement en retard ou en avance dans la solution courante. L'idée est de permettre aux opérations des tâches p et q d'être décalées vers la gauche ou vers la droite pour diminuer respectivement leurs coûts de retard ou d'avance.

D'autres méthodes heuristiques telles que des algorithmes génétiques [153] et une approche multiagent [143] ont été essayées.

Nous avons connaissance de deux algorithmes complets, c'est-à-dire qui fournissent une borne inférieure sur la fonction objectif et, *in fine*, une preuve d'optimalité de la solution produite. Le premier algorithme [13, 15] combine programmation par contraintes et programmation linéaire et est fondé sur l'algorithme de *probe backtrack search*[1] [128]. Il résout d'abord par *probe backtrack search* le sous-problème des opérations qui interviennent directement dans la fonction de coût, c'est-à-dire la dernière opération de chaque tâche (CRS : *Cost Relevant Subproblem*), puis essaie d'étendre par programmation par contraintes la solution de ce sous-problème à une solution du problème complet. Si aucune extension n'est trouvée, alors un branchement est effectué en se servant de la relaxation linéaire comme dans *probe backtrack search* pour choisir la variable de branchement, et le processus est itéré. Une analyse détaillée des expériences numériques [44] montre que cette approche est très efficace lorsque la solution du CRS défini au nœud racine peut être étendue en une solution complète. En effet, le sous-problème CRS est de taille assez petite pour être résolu efficacement par *probe backtrack search* et la fixation des dates de début de la dernière opération de chaque tâche permet souvent de contraindre le problème suffisamment pour guider la programmation par contraintes dans la deuxième phase et trouver rapidement une extension de la solution du CRS (si elle existe). Ce cas de figure correspond à une instance très peu contrainte, ou plus précisément dont les seules contraintes de ressources qui posent éventuellement problème concernent la dernière opération de chaque tâche. Plus de la moitié des instances générées aléatoirement pour tester HLS et CRS [13, 14, 15, 44, 42] exhibent cette propriété, mais de telles instances ne semblent pas réalistes. Au contraire, si la solution du CRS défini au nœud racine ne peut pas être étendue en une solution complète, cette approche n'est pas efficace [44].

Le deuxième algorithme complet [11] est une relaxation lagrangienne qui traite le problème un peu plus général où une date échue est affectée à chaque opération et pas seulement à la dernière opération de chaque tâche. Contrairement à d'autres approches [13, 15, 150, 29] qui relâchent les contraintes de ressources, Baptiste et Sourd relâchent les contraintes de précédence ; le problème peut alors être décomposé

[1] *Probe backtrack search* est une stratégie pour choisir la variable de branchement dans un arbre de programmation par contraintes à partir de la relaxation continue du même problème modélisé en PLNE.

en m sous-problèmes à une machine.

4.2 Amélioration et extension du modèle disjonctif

4.2.1 Le modèle disjonctif

Le modèle disjonctif de programmation linéaire en nombres entiers utilisé notamment par Applegate et Cook [7] s'écrit comme suit. Soit x_{ji} la variable continue qui représente la date de début de l'exécution de la tâche j sur la machine i. La minimisation de la somme des coûts d'avance et retard est exprimée par $\min \sum_{j\in J} z_j$ où, pour tout $j \in J$, la variable z_j représente le coût encouru par la tâche j :

$$z_j \geq \alpha_j(d_j - x_{j\sigma_{jm}} - p_{j\sigma_{jm}}) \tag{4.1}$$

$$z_j \geq \beta_j(x_{j\sigma_{jm}} + p_{j\sigma_{jm}} - d_j) \tag{4.2}$$

Les contraintes de disponibilité sont modélisées par :

$$x_{j\sigma_{j1}} \geq r_j, \ \forall j \in J \tag{4.3}$$

Les contraintes de précédence entre les opérations de chaque tâche sont modélisées par :

$$x_{j\sigma_{j,i+1}} \geq x_{j\sigma_{ji}} + p_{j\sigma_{ji}}, \ \forall j \in J, \ \forall i \in \{1,\ldots,m-1\} \tag{4.4}$$

Les contraintes de ressources sont modélisées de la manière suivante :

$\forall\ p < q \in J, \forall\ i \in \{1,\ldots,m\}$,

$$x_{pi} \geq x_{qi} + p_{qi} - M^i_{pq} y^i_{pq} \tag{4.5}$$

$$x_{qi} \geq x_{pi} + p_{pi} - M^i_{qp} y^i_{qp} \tag{4.6}$$

$$y^i_{pq} + y^i_{qp} = 1 \tag{4.7}$$

$$y^i_{pq} \in \{0,1\} \text{ et } y^i_{qp} \in \{0,1\}$$

L'interprétation est que $y^i_{pq} = 1$ si la tâche p est ordonnancée avant la tâche q sur la machine i, et $y^i_{pq} = 0$ sinon. Nous n'effectuons pas le changement de variables $y^i_{qp} = 1 - y^i_{pq}$ car nous présenterons à la Section 4.2.4 une version différente de la Contrainte 4.7 pour traiter un cas plus général.

Dans l'équation 4.5, M^i_{pq} est un coefficient constant qui majore la valeur de l'expression $x_{qi} + p_{qi} - x_{pi}$ pour tout ordonnancement optimal. En d'autres termes, M^i_{pq} doit être un majorant de la différence entre la date de fin de la tâche q sur la machine i et la date de début de la tâche p sur la machine i. De même, M^i_{qp} doit être une borne supérieure pour $x_{pi} + p_{pi} - x_{qi}$. Par exemple, il est possible de choisir la même valeur pour tous les coefficients M, en général une grande constante arbitraire telle que l'horizon de l'ordonnancement. Il est également possible de calculer plus finement une bonne valeur pour les coefficients M (voir par exemple [48] dans le cas de l'optimisation du *makespan*).

Comme nous l'avons déjà brièvement mentionné à la section précédente, ce modèle a l'avantage d'être simple mais, à cause des coefficients M, il ne lie que faiblement les variables de décision y qui apparaissent dans les contraintes de ressources et les variables auxiliaires x qui apparaissent dans les contraintes de précédence. On s'attend donc à ce que la relaxation linéaire de ce modèle soit de mauvaise qualité, ou qu'il soit difficile d'obtenir de bonnes solutions entières avec ce modèle, comme nous l'avons montré dans [44].

Ce modèle peut être amélioré en ajoutant des coupes, c'est-à-dire des inégalités qui n'excluent aucune solution entière mais renforcent la relaxation pour qu'elle soit une meilleure approximation de l'enveloppe convexe des solutions entières du problème. C'est l'approche suivie par Applegate et Cook [7].

Une autre manière de renforcer cette formulation est d'essayer de calculer des valeurs plus intéressantes pour les coefficients M qui apparaissent dans les contraintes de ressources. En effet, il est intéressant de choisir la plus petite valeur possible pour chaque coefficient M^i_{pq} de façon à resserrer le lien entre les variables x et y. Dans les deux sections suivantes, nous présenterons deux manières différentes de choisir une valeur pour chaque coefficient M^i_{pq}.

Auparavant, notons la propriété suivante du modèle disjonctif, qui nous sera utile dans le calcul d'une valeur intéressante pour chaque M^i_{pq}.

Proposition 4.1 *Si les données temporelles (date de disponibilité, date échue et temps d'exécution de chaque opération) de l'instance à résoudre sont entières pour chaque tâche, alors il existe une solution optimale pour laquelle les dates de début de chaque opération sont entières.*

Preuve[2]. Supposons que les variables de décision y ont été fixées à des valeurs entières correspondant à une solution réalisable et considérons le problème linéaire restant défini par les contraintes 4.1 à 4.6 sur les variables x et z. Chaque point extrême de ce polyèdre correspond à une solution où chaque tâche termine soit à l'heure, soit avant, soit après sa date échue. Donc, c'est aussi un point extrême du polyèdre défini par la matrice où une seule des deux contraintes 4.1 et 4.2 est active pour chaque tâche $j \in J$. (Pour les tâches à l'heure, on conserve la contrainte 4.2 transformée en égalité). Pour tout $j \in J$, effectuons un changement d'échelle pour la variable z_j, selon un facteur α_j si la tâche j est en avance et selon un facteur β_j si la tâche j est à l'heure ou en retard. Le coût encouru pour la tâche j peut alors s'exprimer par $\alpha_j t_j$ ou $\beta_j t_j$, où la variable t_j représente la durée de l'avance ou du retard de la tâche j et est contrainte par l'une des inégalités suivantes :

$$t_j \geq d_j - x_{j\sigma_{jm}} - p_{j\sigma_{jm}} \tag{4.8}$$

$$t_j \geq x_{j\sigma_{jm}} + p_{j\sigma_{jm}} - d_j \tag{4.9}$$

qui remplace respectivement la contrainte 4.1 ou la contrainte 4.2. La matrice des contraintes du problème linéaire ainsi défini est totalement unimodulaire parce qu'elle

[2]Cette preuve est légèrement différente de la preuve donnée par El Sakkout et Wallace [128].

remplit les conditions suffisantes suivantes [133], également utilisées par El Sakkout et Wallace [128] :

1. Tous ses coefficients valent 0, 1 ou -1.
2. Chaque ligne comprend au plus deux coefficients non nuls.
3. L'ensemble des colonnes de la matrice peut être partitionné en deux sous-ensembles :

$$\begin{aligned} S_1 &= \{x_{ji}\}_{j\in J, i\in\{1,\ldots,m\}} \cup \{z_j\}_{\{j\in J|\text{ la tâche } j \text{ est en retard ou à l'heure}\}} \\ S_2 &= \{z_j\}_{\{j\in J|\text{ la tâche } j \text{ est en avance}\}} \end{aligned}$$

qui vérifient les propriétés suivantes :

(a) Si une ligne contient deux coefficients non nuls de même signe, alors l'une des variables correspondantes appartient à S_1 et l'autre appartient à S_2.

(b) Si une ligne contient deux coefficients non nuls de signe opposé, alors les deux variables correspondantes appartiennent au même sous-ensemble.

La matrice étant totalement unimodulaire, si le membre de droite des contraintes est entier, alors les variables de début de toutes les opérations sont entières pour tout point extrême de ce polyèdre. Tout point extrême du polyèdre d'origine, autrement dit toute solution de base du problème d'origine (en particulier la solution optimale), a donc des valeurs entières pour les dates de début de chaque opération. □

Nous supposerons dans toute la suite que les hypothèses de la proposition 4.1 sur l'intégralité des données temporelles sont vérifiées.

Corollaire 4.1 *Le choix de la valeur de chaque M^i_{pq} peut être limité aux valeurs entières.*

4.2.2 Réduction des coefficients à partir d'une solution réalisable

Calcul des coefficients

Nous présentons ici plusieurs manières de calculer les coefficients M^i_{pq} selon différents aspects du problème.

Une première manière naïve de calculer une valeur pour les coefficients M est la suivante.

Proposition 4.2 *Nous pouvons choisir pour chaque coefficient M^i_{pq} la même valeur M calculée de la manière suivante :*

$$M = \max_{j\in J} d_j + \sum_{j\in J}\sum_{i=1}^{m} p_{ji} - \min_{j\in J} r_j \tag{4.10}$$

Preuve. En effet, si l'on choisit la même valeur M pour tous les coefficients M^i_{pq}, alors cette valeur doit être un majorant de la durée totale de l'ordonnancement optimal. Considérons un ordonnancement réalisable quelconque s. La première activité

de s commence nécessairement après la plus petite date de réalisabilité $\min_{j\in J} r_j$. Si s termine après $\max_{j\in J} d_j + \sum_{j\in J}\sum_{i=1}^{m} p_{ji}$, il est possible de construire à partir de s un ordonnancement s_1 de coût inférieur ou égal au coût de s et qui termine avant $\max_{j\in J} d_j + \sum_{j\in J}\sum_{i=1}^{m} p_{ji}$ de la manière suivante. Toute opération qui commence avant $\max_{j\in J} d_j$ dans s garde la même date de début dans s_1. Les opérations qui commencent après $\max_{j\in J} d_j$ dans s sont ordonnancées au plus tôt dans s_1, en conservant sur chaque machine la même séquence des opérations que dans s_1. L'ordonnancement s_1 termine avant $\max_{j\in J} d_j + \sum_{j\in J}\sum_{i=1}^{m} p_{ji}$ car $\sum_{j\in J}\sum_{i=1}^{m} p_{ji}$ est une borne supérieure sur la durée totale de tout ordonnancement (donc en particulier, de tout ordonnancement d'un sous-ensemble d'opérations), quel que soit l'ordre dans lequel les opérations sont exécutées. De plus, l'ordonnancement s_1 a coût inférieur ou égal à celui de s (même coût d'avance, coût de retard inférieur ou égal). Donc, on peut choisir chaque coefficient M comme dans l'équation 4.10 sans exclure l'ordonnancement optimal. □

Étant donné un ordonnancement réalisable s^* qui se termine en $makespan(s^*)$, une amélioration de la Proposition 4.2 serait de choisir :

$$M = \max_{j\in J} d_j + makespan(s^*) - \min_{j\in J} r_j \tag{4.11}$$

Mais, pour pouvoir effectuer cette réduction des coefficients, il faudrait que l'ordonnancement optimal pour le *makespan* s_{MS} et l'ordonnancement optimal pour le coût d'avance et retard s_{ET} vérifient la propriété suivante :

$$makespan(s_{ET}) \leq \max_{j\in J} d_j + makespan(s_{MS}) \tag{4.12}$$

Or, cette propriété est fausse dans le cas $d_j = 0\ \forall j \in J$, comme le montre le contre-exemple de la Figure 4.1. En effet, les séquences des opérations sur chaque machine ne sont pas nécessairement identiques dans s_{ET} et s_{MS}. Il reste à déterminer si cette propriété est vraie lorsque $d_j \geq r_j + \sum_{i=1}^{m} p_{ji}\ \forall j \in J$, autrement dit, lorsque chaque tâche prise individuellement peut finir en avance ou à l'heure (ce que l'on pourrait supposer sans une trop grande perte de généralité). En l'absence d'une telle preuve[3], nous nous somme donc contentés pour l'instant de l'Équation 4.10.

Rappelons que tous les coefficients M ne doivent pas nécessairement prendre la même valeur. M^i_{pq} doit être un majorant de $x_{qi} + p_{qi} - x_{pi}$ uniquement pour le triplet (i, p, q) considéré. Cette remarque nous permet de calculer une meilleure valeur :

$$M^i_{pq} = \max_{j\in J} d_j + \sum_{j\in J}\sum_{i=1}^{m} p_{ji} - E_{pi} - F_{qi} \tag{4.13}$$

où E_{pi} est la date de début au plus tôt de la tâche p sur la machine i (la date de disponibilité de p plus la somme des durées d'exécution de p sur les machines qui viennent avant i dans l'ordre prescrit pour la tâche p), et F_{qi} est le temps nécessaire à l'accomplissement de la tâche q après son exécution sur la machine i (la somme des durées

[3] La preuve présentée dans [39] comporte une erreur.

Tâche	Permutation des machines	Durée des opérations	Date échue	Coût de retard
A	$\sigma_A = \{1,2\}$	$p_{A1} = p_{A2} = 2$	$d_A = 0$	$\beta_A = 1$
B	$\sigma_B = \{1,2\}$	$p_{B1} = p_{B2} = 1$	$d_B = 0$	$\beta_B = 1$
C	$\sigma_C = \{2,1\}$	$p_{C1} = 2,\ p_{C2} = 1$	$d_C = 0$	$\beta_C = 2$

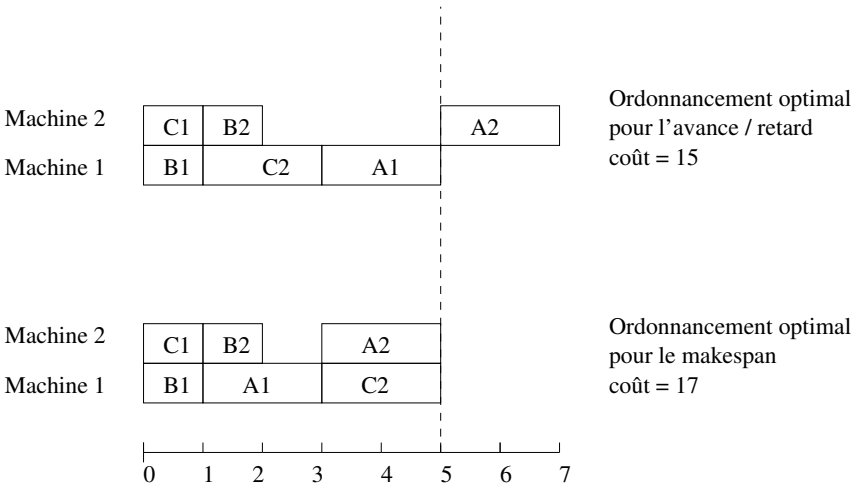

FIG. 4.1 – Un contre-exemple pour l'équation 4.12.

d'exécution de q sur les machines restantes) :

$$\begin{aligned} E_{pi} &= r_p + \sum_{a=1}^{\sigma_{pi}^{-1}-1} p_{p\sigma_{pa}} \\ F_{qi} &= \sum_{a=\sigma_{qi}^{-1}+1}^{m} p_{q\sigma_{qa}} \end{aligned}$$

où σ_{pi}^{-1} est la position de la ressource i dans l'ordre prescrit pour la tâche p.

Remarquons que M_{pq}^i doit en fait seulement être un majorant des valeurs possibles pour $x_{qi} + p_{qi} - x_{pi}$ dans les ordonnancements de coût inférieur au coût z^* de l'ordonnancement courant. Dans toute solution améliorante, le coût de retard de la tâche q est inférieur à z^*, donc la date de fin de la tâche q vérifie $C_q \leq d_q + \frac{z^*}{\beta_q}$. Par définition de F, $x_{qi} + p_{qi} + F_{qi} \leq C_q$, et par définition de E, $x_{pi} \geq E_{pi}$. En appliquant de plus le corollaire 4.1, nous obtenons donc finalement l'équation suivante qui définit une nouvelle valeur pour M_{pq}^i :

$$x_{qi} + p_{qi} - x_{pi} \leq d_q + \left\lfloor \frac{z^*}{\beta_q} \right\rfloor - F_{qi} - E_{pi} \tag{4.14}$$

Notons également que, dans toute solution améliorante, la somme du coût d'avance de la tâche q et du coût de retard de la tâche p est inférieure à z^* : $\beta_q(C_q - d_q)^+ + \alpha_p(d_p - C_p)^+ \leq z^*$. Notons $\gamma_{pq} = \min(\beta_q, \alpha_p)$. L'inégalité précédente implique $\gamma_{pq}(C_q - d_q + d_p - C_p) \leq z^*$. Si i est la dernière machine sur laquelle p doit être exécutée, alors $x_{pi} + p_{pi} = C_p$. En utilisant comme précédemment la définition de F_{qi} et le corollaire 4.1, nous obtenons une nouvelle inégalité pour définir M_{pq}^i dans ce cas particulier :

$$x_{qi} + p_{qi} - x_{pi} \leq d_q + \left\lfloor \frac{z^*}{\gamma_{pq}} \right\rfloor - F_{qi} - d_p + p_{pi} \tag{4.15}$$

Il est à noter qu'il n'existe pas de relation de dominance entre les Équations 4.13 à 4.15. Selon les caractéristiques numériques de la solution s^* et de l'instance à résoudre, la meilleure valeur pour M_{pq}^i peut être obtenue par l'Équation 4.13 ou l'Équation 4.14 (ou l'Équation 4.15 si les conditions de ce cas particulier sont remplies). Nous prenons donc systématiquement la plus petite des deux (ou trois) valeurs obtenues.

Notons enfin que si l'une des Équations 4.13 à 4.15 produit une valeur négative ou nulle pour M_{pq}^i, alors la tâche p est nécessairement exécutée après la tâche q sur la machine i et on peut directement fixer $y_{pq}^i = 0$ et $y_{qp}^i = 1$.

Mise en œuvre pratique

Les Équations 4.13 à 4.15 peuvent être mises en œuvre de trois manières différentes. Elles permettent d'abord de calculer des valeurs initiales pour chaque M_{pq}^i à partir d'une solution construite de manière heuristique, et ainsi de créer un premier modèle de programmation linéaire en nombres entiers.

Ensuite, lorsque des solutions améliorantes sont découvertes pendant l'optimisation de ce premier modèle, il est possible de calculer de nouvelles valeurs pour chaque M_{pq}^i

et d'ajouter des coupes correspondant aux contraintes de ressource mises à jour, ou bien de construire un nouveau modèle renforcé et de recommencer l'optimisation à partir de zéro. Ajouter régulièrement des coupes présente l'inconvénient de faire augmenter rapidement la taille du modèle. De plus, si les solutions entières découvertes ne sont pas significativement meilleures, il est peu probable que leur introduction apporte beaucoup à la résolution du problème. Au contraire, si la qualité des solutions entières s'améliore considérablement, alors il peut être intéressant de reprendre l'exploration de l'arbre de *branch-and-cut* à partir de zéro en prenant des décisions de branchement en haut de l'arbre meilleures que celles prises précédemment. C'est pourquoi nous avons préféré expérimenter cette deuxième résolution itérative.

Un premier modèle est construit avec les coefficients M^i_{pq} calculés à partir d'une première solution réalisable s fournie par un algorithme extérieur. Ce modèle est utilisé jusqu'à ce qu'au moins f nœuds aient été explorés et qu'une solution s' de coût inférieur de p% au moins au coût de s ait été trouvée. Un nouveau modèle est alors créé avec les coefficients M^i_{pq} calculés à partir de s'. L'optimisation reprend avec ce nouveau modèle, avec le même critère d'arrêt que précédemment, et ainsi de suite. Plusieurs modèles peuvent donc être construits successivement. La seule information transmise du modèle k au modèle $k+1$ est la meilleure solution trouvée pendant l'optimisation du modèle k qui est utilisée comme première solution pour le modèle $k+1$. Les paramètres p et f permettent d'arbitrer entre d'une part le gain procuré par des coefficients M^i_{pq} plus petits et une probable meilleure stratégie de traversée de l'arbre de *branch-and-cut*, et d'autre part le risque de perdre du temps en explorant plusieurs fois les mêmes régions de l'espace de recherche pendant la résolution du modèle k et du modèle $k+1$.

4.2.3 Réduction heuristique des coefficients (MCORE)

Algorithme

Les approches que nous avons proposées dans la section précédente pour calculer des valeurs plus serrées pour les coefficients M sont exactes. En effet, elles n'excluent aucune solution entière améliorante et elles permettent de prouver l'optimalité des solutions obtenues. Cependant, il est également possible de diminuer heuristiquement la valeur de chaque coefficient M^i_{pq} de façon à trouver plus facilement de bonnes solutions entières grâce à ce modèle renforcé, mais sans garantir que cette réduction de l'espace de recherche n'exclue aucune solution améliorante. Afin de conserver la possibilité de prouver l'optimalité, cette procédure est à utiliser comme heuristique de *branch-and-cut*, en marge de l'exploration arborescente du modèle exact.

Une manière intuitive de définir un voisinage autour d'un ordonnancement s^* est de considérer tous les ordonnancements où, pour chaque opération, la date de début n'est pas très différente de la date de début de cette opération dans s^*. L'idée est donc de resserrer les bornes de chaque variable x_{pi} autour de sa valeur x^*_{pi} dans la solution

courante s^*. Mais, avec de tels changements de bornes, un nouveau majorant peut être calculé sur chacune des expressions $x_{qi} + p_{qi} - x_{pi}$, ce qui produit une valeur plus petite pour chaque coefficient M^i_{pq}.

Plus précisément, notre heuristique MCORE (*big-M COefficient REduction*) définit le voisinage comme l'ensemble des solutions du modèle disjonctif où les coefficients sont définis de la manière suivante[4] :

$$M^i_{pq} = M^i_{qp} = 2 * \max\left(x^*_{qi} + p_{qi} - x^*_{pi}, x^*_{pi} + p_{pi} - x^*_{qi}\right) \tag{4.16}$$

et où les bornes des variables qui représentent les dates de début des opérations sont spécifiées par :

$$x^*_{pi} - \min_{q \in J} M^i_{pq} \leq x_{pi} \leq x^*_{pi} + \min_{q \in J} M^i_{pq} \tag{4.17}$$

Le coefficient 2 dans l'équation 4.16 a été choisi empiriquement. Il procure un degré de liberté assez important pour définir un voisinage contenant de nombreuses solutions, tout en réduisant les coefficients assez significativement pour faciliter notablement la résolution du modèle.

Le sous-modèle ainsi défini est résolu par *branch-and-cut*. Même si la réduction des coefficients rend le sous-modèle plus facile que le modèle d'origine, il n'est souvent pas possible de le résoudre à l'optimalité en un temps raisonnable. Donc, comme pour RINS et le *local branching*, l'exploration du sous-modèle est tronquée en imposant une limite *nl* sur le nombre maximum de nœuds. Lorsque cette limite est atteinte, la solution entière courante du modèle global est mise à jour avec la meilleure solution entière trouvée dans le sous-modèle (le cas échéant), et l'exploration de l'arbre global reprend. Plus précisément, il est clair que toute solution entière du sous-modèle est également une solution du modèle d'origine. Mais, lorsque la séquence des opérations sur chaque machine est fixée, le placement dans le temps des opérations qui est optimal pour le sous-modèle n'est pas nécessairement optimal pour le modèle d'origine. Donc, seules les valeurs des variables y sont transmises du sous-modèle au modèle d'origine, puis ces variables étant fixées, la relaxation linéaire du modèle d'origine est résolue pour déterminer la valeur des variables x. On observe que la solution ainsi obtenue est souvent légèrement meilleure que la solution trouvée pendant l'exploration du sous-modèle.

MCORE peut être appelé chaque fois qu'une nouvelle solution entière est découverte dans l'arbre de recherche initial. En pratique, cette heuristique est coûteuse, en particulier souvent plus coûteuse que RINS, donc nous l'appelons seulement tous les f nœuds et si la solution entière courante est strictement meilleure que la solution utilisée par le dernier appel de MCORE.

[4] M^i_{pq} prend en fait la plus petite valeur parmi les valeurs produites par les Équations 4.13 à 4.15 et la valeur heuristique proposée par l'Équation 4.16. Il est à noter que les Équations 4.14 et 4.15 sont utilisées avec la somme des coûts d'avance et retard atteinte dans la solution courante s^*, ce qui permet déjà d'obtenir une réduction (exacte) du coefficient M^i_{pq} si s^* est de meilleure qualité que la solution réalisable utilisée pour construire le modèle d'origine.

Relation avec les algorithmes de la littérature

La définition du voisinage MCORE est intrinsèquement différente des approches à base de *large neighborhood search* (LNS) telles que l'heuristique de *relaxation induced neighborhood search* (RINS) présentée au chapitre précédent. En effet, les approches à base de LNS fixent une partie des variables de décision (ici les variables disjonctives y) à leur valeur dans la solution courante, puis considèrent le modèle initial sur le reste des variables. L'avantage de ce type d'approche est que le sous-modèle définissant le voisinage comporte moins de variables que le modèle d'origine. L'inconvénient dans le cas particulier que nous étudions ici est que le sous-modèle a la même structure que le modèle d'origine : il comporte toujours des contraintes de type *big-M*, avec des coefficients inchangés, donc il est toujours difficile à résoudre. Au contraire, MCORE ne fixe aucune variable de décision mais définit un sous-modèle qui comporte le même nombre de variables que le modèle d'origine et c'est la réduction des coefficients de type *big-M* qui rend ce sous-modèle plus facile à résoudre que le modèle d'origine.

Plus généralement, MCORE se distingue des heuristiques existantes pour le *branch-and-cut* parce qu'elle exploite la valeur des variables *continues* dans la solution courante. En effet, les heuristiques pour le *branch-and-cut* que nous avons présentées dans la Section 1.3 du Chapitre 1, en particulier les heuristiques d'arrondis et les heuristiques de réparation de solutions entières irréalisables, opèrent uniquement sur les variables entières pour se déplacer de solution en solution voisine. La valeur des variables continues est simplement déterminée en résolvant la relaxation linéaire du modèle, un certain nombre de variables entières ayant été fixées. De manière similaire, RINS et le *local branching* définissent le voisinage de la solution courante en ajoutant des contraintes portant uniquement sur les variables entières, puis les valeurs des variables continues sont obtenues au cours de l'exploration du voisinage par *branch-and-cut*. Les variables continues sont donc considérées comme des variables secondaires parce qu'il est possible de les déterminer en résolvant un simple programme linéaire une fois les variables entières fixées. Mais, on peut aussi considérer qu'inversement, si les variables continues sont fixées, alors les variables entières deviennent fortement contraintes et il est plus facile de déterminer leur valeur. Dans le cas du modèle disjonctif, si l'on connaît les dates de début x de chaque opération, alors les valeurs des variables disjonctives y peuvent être déterminées immédiatement. Donc, cela a un sens d'exploiter uniquement les variables continues pour définir un voisinage de la solution courante.

Notons enfin que MCORE est un algorithme *conceptuellement hybride*. En effet, il utilise les trois concepts de recherche locale (voisinage, intensification, diversification) mais les transpose dans le cadre de la programmation linéaire en nombres entiers et seule la technique de *branch-and-cut* est utilisée. Le voisinage est défini par un sous-modèle qui exploite les propriétés des contraintes de type *big-M* qui sont particulières à la modélisation en PLNE. L'intensification est obtenue en explorant le voisinage par *branch-and-cut*. La diversification est obtenue par les solutions découvertes pendant la résolution par *branch-and-cut* du modèle d'origine.

Généralisation

L'heuristique MCORE est particulièrement adaptée au modèle disjonctif pour le problème d'ordonnancement d'atelier parce que les contraintes de type *big-M* représentent un aspect important de ce problème et parce que les variables continues x suffisent pour définir un ordonnancement de manière univoque. Mais il serait tout à fait possible d'appliquer MCORE à n'importe quel modèle de programmation linéaire en nombres entiers qui comporte des contraintes de type *big-M*. Le sous-modèle est défini en remplaçant chaque contrainte de type *big-M* où les variables x_i sont continues et la variable y est binaire :

$$\sum_{i \in I} a_i x_i \leq My$$

par une contrainte où le coefficient de type *big-M* a été réduit heuristiquement d'après la solution entière courante x^* :

$$\sum_{i \in I} a_i x_i \leq my \text{ avec } m = 2 \sum_{i \in I} a_i x_i^*$$

et les bornes des variables sont modifiées de la manière suivante :

$$x_i^* - \frac{m}{|a_i|} \leq x_i \leq x_i^* + \frac{m}{|a_i|}, \forall i \in I$$

Notons que la généralisation que nous venons de présenter ne correspond pas tout à fait à la forme particulière de MCORE que nous avons appliquée au problème d'ordonnancement avec coûts d'avance et retard. En effet, si l'on applique littéralement la forme générale de MCORE à ce problème particulier, c'est-à-dire en choisissant $m^i_{pq} = 2(x^*_{qi} + p_{qi} - x^*_{pi})$ et $m^i_{qp} = 2(x^*_{pi} + p_{pi} - x^*_{qi})$, alors l'un des coefficients m^i_{pq} ou m^i_{qp} sera négatif, ce qui fixera l'ordre entre la tâche p et la tâche q sur la machine i comme dans la solution courante, donc le sous-problème ne contiendra pas d'autre solution que la solution courante. La forme particulière de MCORE que nous avons présentée précédemment est nécessaire ici dans la mesure où les deux variables liées très fortement par la contrainte 4.7 (ou lorsqu'on autorise la non-allocation d'activités, par la contrainte 4.18 présentée à la section suivante) apparaissent dans deux contraintes différentes de type *big-M*. Il reste à vérifier sur d'autres modèles si notre proposition de généralisation de MCORE peut s'appliquer telle quelle, ou s'il faut l'adapter de manière légèrement différente pour chaque modèle en n'en retenant que le principe.

4.2.4 Autoriser les opérations non-allouées

Dans le jeu de données MaScLib [109] que nous utiliserons pour nos expériences rapportées à la Section 4.3, certaines opérations peuvent être non-allouées (*unperformed*). Comme pour les autres opérations, il faut associer une date de début à une opération qui est non-allouée, son temps d'exécution sera le même que si elle était allouée et cette opération apparaîtra comme d'habitude dans le graphe de précédence. Par contre, une opération non-allouée ne requiert de capacité sur aucune machine, mais le fait

d'être non-allouée implique un coût supplémentaire. Cette situation correspond en pratique à la sous-traitance d'une opération ou à l'allocation de capacité supplémentaire sur une machine pour exécuter cette opération. Nous modélisons cette possibilité en introduisant la variable binaire u^i_j qui vaut 1 si l'opération σ^{-1}_{ji} de la tâche j est non-allouée, et $u^i_j = 0$ sinon. La fonction objectif devient donc :

$$\min \sum_{j \in J} z_j + \sum_{j \in J} \sum_{i=1}^{m} unp^i_j u^i_j$$

où unp^i_j est le coût supplémentaire encouru lorsque l'opération σ^{-1}_{ji} de la tâche j est non-allouée. Pour lier les contraintes de ressources aux variables de non-allocation, l'équation 4.7 est remplacée par l'équation suivante :

$$y^i_{pq} + y^i_{qp} \leq 1 + u^i_p + u^i_q \tag{4.18}$$

Les différentes manières de réduire les coefficients M^i_{pq} que nous avons présentées à la Section 4.2.2 restent valables dans le cas où des opérations peuvent être non-allouées. En effet, elles ne font intervenir que les contraintes de précédence entre opérations et les considérations sur les dates échues, qui sont prises en compte même en cas de non-allocation. La réduction heuristique des coefficients M^i_{pq} présentée à la Section 4.2.3 reste également applicable dans la mesure où une date de début doit être affectée à toute opération, même non-allouée.

4.2.5 Supprimer les symétries

Certaines instances du jeu de données MaScLib [109] que nous avons utilisé dans nos expériences comportent de nombreuses tâches identiques (même disponibilité, même date échue et mêmes durées d'exécutions sur la même permutation de machines). En effet, il est courant dans la pratique de fabriquer des lots de produits identiques. Le modèle comporte alors un grand nombre de symétries qu'il est intéressant de supprimer pour accélérer la résolution.

Considérons un ensemble $S = \{1, \ldots, s\}$ de tâches identiques. Dans le cas où la non-allocation d'opérations n'est pas autorisée, nous ajoutons les contraintes suivantes pour imposer des précédences entre des opérations identiques :

$$y^i_{p,p+1} = 1 \text{ et } y^i_{p+1,p} = 0, \ \forall p \in \{1, \ldots, s-1\}, \ \forall i \in \{1, \ldots, m\}$$

Dans le cas où la non-allocation est autorisée, nous ajoutons les contraintes suivantes :

$$y^{m_1}_{p,p+1} = 1, \quad \forall p \in \{1, \ldots, s-1\} \tag{4.19}$$

$$u^{m1}_p \leq u^{m1}_{p+1}, \quad \forall p \in \{1, \ldots, s-1\} \tag{4.20}$$

où $m_1 = \sigma^{-1}_{11} = \sigma^{-1}_{21} = \ldots = \sigma^{-1}_{s1}$ est la première machine sur laquelle ces tâches doivent être exécutées. La première famille d'inégalités 4.19 permet de fixer la précédence entre deux opérations identiques lorsqu'elles sont toutes les deux allouées. La

deuxième famille d'inégalités 4.20 permet d'ordonner un ensemble d'opérations identiques selon le critère de non-allocation. Contrairement au cas où la non-allocation n'est pas autorisée, seules les symétries entre chaque première opération d'un sous-ensemble de tâches identiques sont considérées, car il n'est pas trivial de supprimer les symétries en prenant en compte simultanément les précédences et la non-allocation (par exemple, on ne peut pas imposer brutalement qu'une opération non-allouée commence avant une opération allouée identique).

Notons dans les deux cas que le nombre de contraintes ainsi ajoutées est linéaire.

4.3 Résultats expérimentaux

4.3.1 Le jeu de données MaScLib

Pour mesurer la performance des variantes de modélisation et de résolution présentées à la section précédente, nous avons utilisé le jeu de données MaScLib [109] qui regroupe pour différents problèmes d'ordonnancement des instances construites à partir de données industrielles et d'instances académiques difficiles. Nous nous sommes concentrés sur la série des problèmes NCGS « No-Calendar General-Shop » qui constituent une légère généralisation du problème d'ordonnancement d'atelier défini à la Section 4.1.2. Dans la suite, sauf mention contraire, nous donnerons des résultats sur toutes les instances de la série NCGS sauf les instances NCGS_75, NCGS_75a (1250 opérations), NCGS_81 et NCGS_81a (2500 opérations). La programmation linéaire en nombres entiers est capable de traiter ces quatre instances de très grande taille, mais la limite de temps de calcul préconisée par MaScLib ne permet pas de trouver une première solution entière sur certaines instances et, dans tous les cas, ne permet pas d'explorer un nombre suffisant de nœuds dans l'arbre de *branch-and-cut* pour que les méthodes présentées dans la section précédente produisent une différence significative de performance par rapport au modèle le plus naïf.

Chaque instance est successivement optimisée avec les quatre critères suivants : somme des coûts d'avance et de retard (ET), somme des coûts de retard (T), somme des coûts d'avance, de retard et de non-allocation (ET_UNP), somme des coûts de retard et de non-allocation (T_UNP). Notons que MaSclib ne contient pas seulement des instances de type NCGS, mais aussi des instances dans lesquelles des opérations peuvent être exécutées selon différents modes (ressources alternatives ou différentes manières d'exécuter une opération sur la même ressource), chaque mode ayant un coût potentiellement différent. Aussi chaque instance de MaScLib inclut-elle un coût de mode (*mode cost*) à prendre en compte pour chaque opération dans le coût total à optimiser. Dans le cas NCGS, chaque opération ne peut être exécutée que selon un seul mode, donc ce coût de mode est constant, mais il est tout de même pris en compte dans les résultats. Ce coût de mode est encouru même lorsqu'une opération est non-allouée, le coût de non-allocation représentant en fait le surcoût par rapport à l'allocation de

l'opération. Le coût de mode constitue donc ici une borne inférieure triviale sur le coût de toute solution réalisable. Nous l'utiliserons à la Section 4.3.4 pour évaluer la qualité des bornes inférieures produites par la programmation linéaire en nombres entiers.

Instance	Taille	Temps	Coût de mode	Critère d'optimisation			
				ET	T	ET_UNP	T_UNP
NCGS_12a	24 / 2	150	2400	**3911**	*2476*	**3911 (d)**	*2476*
NCGS_12	24 / 2	150	2400	**4414**	*2552*	**4414**	*2552*
NCGS_13a	24 / 2	150	2400	**2786***	*2552*	**2786***	*2552*
NCGS_13	24 / 2	150	2400	**2975***	*2704*	**2975**	*2704*
NCGS_21a	60 / 5	300	6000	**274300**	6200	**265750**	6200
NCGS_21	60 / 5	300	5700	**24697 (d)**	5708	**24697**	5708
NCGS_31a	75 / 5	450	7500	**405200**	65450	**355000**	**43500**
NCGS_31	75/ 5	450	6675	**32651**	**15174**	**32651**	**14847**
NCGS_41a	100 / 10	600	10000	**11070**	10360	**10300**	**10260**
NCGS_41	100 / 10	600	10000	**22510**	15760	**12990**	**12000**
NCGS_45a	150 / 10	600	15000	**18330**	16590	**15510**	**15370**
NCGS_45	150 / 10	600	15000	45510	20040	**20300**	**18000**
NCGS_51a	200 / 10	1200	20000	24170	20930	**20530**	20100
NCGS_51	200 / 10	1200	20000	33260	24100	**25610**	21000
NCGS_52a	230 / 8	1200	23000	35410	32940	35410 (d)	32940
NCGS_52	230 / 8	1200	23000	113120	96320	97250	80440
NCGS_53a	250 / 8	1200	25000	**37600**	**37580**	**37600 (d)**	**36930**
NCGS_53	250 / 8	1200	25000	**114020**	**111550**	**114020 (d)**	**104060**
NCGS_54a	260 / 10	1200	26000	**41850**	35150	38020	29050
NCGS_54	260 / 10	1200	26000	81600	73000	64250	53750
NCGS_55a	260 / 10	1200	26000	**32000**	27950	**31500**	27250
NCGS_55	260 / 10	1200	26000	42350	35750	42300	35500
NCGS_75a	1250 / 30	2400	125000	153840	146280	153840 (d)	146280 (d)
NCGS_75	1250 / 30	2400	125000	216565	212360	216565 (d)	212360 (d)
NCGS_81a	2500 / 30	3600	250000	333700	289960	333700 (d)	289960 (d)
NCGS_81	2500 / 30	3600	250000	444305	413760	430800 (d)	398540

TAB. 4.1 – Meilleures solutions connues.

Le Tableau 4.1 donne pour chacune des 26 instances de la série NCGS que nous avons étudiées le coût de la meilleure solution connue pour chacun des critères d'optimisation que nous venons d'énumérer. Notons qu'une solution obtenue en optimisant le critère ET ou T (donc où toutes les opérations sont allouées) est également réalisable pour le cas où la non-allocation d'opérations est autorisée et a le même coût pour le critère ET_UNP que pour le critère ET (respectivement T_UNP et T). Inversement, si une solution obtenue en optimisant le critère ET_UNP ou T_UNP ne comporte aucune opération non-allouée alors cette solution est réalisable pour le critère ET ou T. De la même façon, une solution obtenue en optimisant le coût de retard peut également constituer une bonne solution pour le coût d'avance et retard, et vice-versa. Dans le Tableau 4.1, nous signalons par « (d) » les cas où la meilleure solution connue a été obtenue par déduction de la meilleure solution connue pour un autre critère.

Les meilleures solutions énumérées dans le Tableau 4.1 ont été obtenues par l'une de nos variantes du modèle de programmation linéaire en nombres entiers que nous expliciterons à la section suivante (dans 43 cas, indiqués en **gras**), ou (dans 53 cas) par un algorithme hybride CP+LS (programmation par contraintes et recherche locale)

en cours de développement à ILOG destiné à résoudre de manière robuste divers problèmes d'ordonnancement manufacturiers que nous détaillerons brièvement ci-après. Les cas où l'une de nos variantes du modèle de programmation linéaire en nombres entiers et CP+LS obtiennent conjointement la meilleure solution connue sont indiqués en *italique* (8 cas). Les trois instances pour lesquelles l'optimum est connu sont dénotées par une étoile.

Pour chaque instance, le Tableau 4.1 indique également la taille (nombre d'opérations / nombre de ressources), la limite de temps en secondes préconisée par MaScLib pour la résolution sur un ordinateur cadencé à 1 GHz, et le coût de mode total.

4.3.2 Les méthodes comparées

Le Tableau 4.2 indique les différentes versions de la programmation linéaire en nombres entiers que nous avons comparées. Nous avons fait varier :

- Le modèle utilisé :
  - version naïve : le coefficient M est calculé selon la Proposition 4.2 ;
  - version sophistiquée : les coefficients M^i_{pq} calculés selon les Équations 4.13 à 4.15 à partir de la première solution trouvée par l'algorithme CP+LS. La première solution de CP+LS est toujours obtenue en optimisant uniquement le coût de retard (les pénalités d'avance et de non-allocation ne sont pas considérées) parce que c'est le critère qui permet d'obtenir très rapidement une première solution de qualité acceptable pour le calcul des coefficients M^i_{pq}. On impose de même que toutes les opérations soient allouées pour limiter l'espace de recherche de cette première solution. Pour construire la première solution, CP+LS ordonne les opérations au fur et à mesure suivant une combinaison de trois critères : la date de début au plus tôt, la date de fin au plus tard et la date échue de la tâche correspondante propagée pour se rapporter à l'opération considérée. Chaque fois qu'une opération est ordonnancée, son positionnement dans le temps est propagé, ce qui entraîne la mise à jour des dates utilisées pour le choix de l'opération suivante.
- La technique de résolution employée :
  - Stratégie par défaut de CPLEX 9 ;
  - RINS : paramètre `IloCplex::MIPEmphasis=4` de CPLEX 9 ;
  - *Guided dives* : paramètre `IloCplex::DiveType=3` de CPLEX 9 ;
  - Résolution itérative par une série de modèles avec des coefficients M^i_{pq} de plus en plus petits, comme indiqué à la fin de la Section 4.2.2. Nous avons choisi expérimentalement $p = 50\%$ et $nl = 1000$.
  - Heuristique MCORE présentée à la Section 4.2.3 : après quelques essais, nous avons choisi les paramètres $nl = 100$ et $f = 500$.
- L'utilisation ou non de la première solution trouvée par l'algorithme CP+LS comme première solution (*MIP start*) du programme linéaire en nombres entiers.

Nom	Modèle	Stratégie de résolution	Première solution de CP+LS utilisée comme *MIP start* ?
MIP 1	Naïf	Stratégie par défaut	Non
MIP 2	Naïf	RINS	Non
MIP 3	Sophistiqué	Stratégie par défaut	Non
MIP 4	Sophistiqué	Stratégie par défaut	Oui
MIP 5	Sophistiqué	RINS	Oui
MIP 6	Sophistiqué	*Guided dives*	Oui
MIP 7	Sophistiqué	– RINS – *Guided dives*	Oui
MIP 8	Sophistiqué	– RINS – *Guided dives* – Résolution itérative	Oui
MIP 9	Sophistiqué	– RINS – *Guided dives* – Heuristique MCORE	Oui

TAB. 4.2 – Les différentes variantes de programmation linéaire en nombres entiers.

Nous comparerons également les résultats de ces différentes variantes de programmation linéaire en nombres entiers avec les résultats de l'algorithme CP+LS utilisé seul (avec le critère d'optimisation d'origine). CP+LS combine programmation par contraintes et *large neighborhood search*. Á chaque terme de l'objectif (coûts d'avance, retard ou non-allocation) est associée une stratégie pour réduire ce terme tout en limitant l'augmentation des autres termes par rapport à leur valeur dans la solution courante. Les différentes stratégies sont appelées itérativement pour améliorer progressivement l'objectif global.

Toutes les expériences ont été réalisées sur un processeur Intel Xeon cadencé à 2.8 GHz avec le système d'exploitation Linux et 2 Go de mémoire vive. La limite de temps utilisée est la moitié de celle préconisée par MaScLib.

4.3.3 Qualité des solutions réalisables

Nous présentons maintenant les résultats des dix algorithmes détaillés à la section précédente. Nous comparons dans cette section la qualité des solutions réalisables obtenues et nous nous intéresserons dans la section suivante à la qualité des bornes inférieures.

Comme dans le chapitre précédent, la qualité des solutions est exprimée par le ratio entre le coût de la solution obtenue par un algorithme donné et le coût de la meilleure solution connue indiquée dans le Tableau 4.1. Ce chiffre n'est bien sûr jamais inférieur à 1. Étant donné le nombre élevé d'instances, de critères d'optimisation et d'algorithmes ($22 * 4 * 10$), nous présentons les résultats de manière agrégée, en donnant dans le tableau 4.3 la moyenne géométrique du ratio précédemment défini sur toutes les instances, critère par critère. Pour chaque critère, le meilleur ratio est indiqué en gras.

Algorithme	Critère d'optimisation				Moyenne
	ET	T	ET_UNP	T_UNP	générale
MIP 1	1.626	1.708	1.370	1.774	1.612
MIP 2	1.607	1.653	1.304	1.933	1.609
MIP 3	1.656	1.866	1.404	1.733	1.656
MIP 4	1.494	1.415	1.208	1.313	1.353
MIP 5	1.378	1.339	1.144	1.194	1.260
MIP 6	1.269	1.219	1.102	1.124	1.176
MIP 7	1.192	1.188	1.088	1.094	1.140
MIP 8	1.179	1.174	1.105	1.079	1.133
MIP 9	1.130	1.155	**1.060**	**1.056**	1.099
CP+LS	**1.107**	**1.015**	1.148	1.082	**1.087**

TAB. 4.3 – Qualité des solutions réalisables : résultats agrégés par critère d'optimisation.

Avant de détailler les écarts de performance entre les différents algorithmes de programmation linéaire en nombres entiers, il est à remarquer avant toute chose que la meilleure variante de programmation linéaire en nombres entiers (MIP 9) est tout à fait compétitive vis-à-vis de l'algorithme hybride spécifique CP+LS quant à la qualité des solutions réalisables produites[5], en particulier lorsque la non-allocation d'opérations est autorisée.

Considérons maintenant les neuf variantes de programmation linéaire en nombres entiers. La conclusion essentielle est que le travail sur la modélisation et le travail sur la résolution sont complémentaires ; les deux aspects sont indispensables pour obtenir des solutions entières de qualité. En effet, si l'on conserve la stratégie de résolution par défaut, passer d'un modèle naïf (MIP 1) à un modèle plus sophistiqué (MIP 3) ne permet pas d'améliorer la qualité des solutions obtenues. De même, si l'on conserve un modèle naïf, passer de la stratégie de résolution par défaut (MIP 1) à une heuristique puissante de programmation linéaire en nombres entiers (MIP 2) ne permet pas une amélioration significative des performances.

La deuxième conclusion est que la première solution joue également un rôle essentiel, comme le montre la différence de performance entre MIP 3 (pas de première solution) et MIP 4 (première solution fournie par CP+LS). Paradoxalement, comme le montre le Tableau 4.4, la première solution trouvée par CP+LS n'est pas de bonne qualité et elle est même de coût supérieur à la solution trouvée en fin d'optimisation par les variantes MIP 1 à MIP 3 qui n'utilisent pas de première solution extérieure. Mais cette première solution est tout de même significativement meilleure que la première solution trouvée par CPLEX sur les modèles MIP 1 ou MIP 3, et c'est ce qui lui permet d'accélérer la résolution. Ces chiffres montrent qu'une première solution trouvée par une heuristique *ad hoc* est très utile, même si elle est de mauvaise qualité, et même si les techniques de résolution employées sont robustes vis-à-vis de la mauvaise qualité des premières solutions, comme nous l'avons montré pour RINS dans les Sections 3.4.3 et 3.4.4 du chapitre précédent. Ceci est vrai en particulier lorsque le temps de résolution est très limité, comme c'est le cas ici d'après les limites de temps préconisées par MaScLib. Commentons enfin la ligne « MIP 4 » du Tableau 4.4. De la solution trouvée par CP+LS, seules les valeurs des variables binaires (variables disjonctives et variables de non-allocation) sont exploitées par la programmation linéaire en nombres entiers. Une fois ces variables fixées, le placement dans le temps de chaque opération est déterminé optimalement en résolvant la relaxation linéaire du problème, ce qui permet souvent de trouver une meilleure solution que la solution complète produite par CP+LS. Cette amélioration n'est visible que pour les critères ET et ET_UNP, car lorsqu'il n'y a pas de coûts d'avance, le placement optimal est simplement le placement au plus tôt dans le temps et c'est ce qui est réalisé par CP+LS.

Les lignes MIP 5 à MIP 7 du Tableau 4.3 montrent comme nous l'avions déjà

[5]Différents tests statistiques (test de Student, test non-paramétrique du signe) montrent que les performances globales de CP+LS et de MIP 9 ne sont pas significativement différentes.

Algorithme	Critère d'optimisation				Moyenne générale
	ET	T	ET_UNP	T_UNP	
MIP 1	6.798	7.660	6.241	10.536	7.650
MIP 3	6.441	6.252	6.430	10.591	7.237
CP+LS	2.091	**1.535**	2.248	**1.636**	1.853
MIP 4	**1.728**	**1.535**	**1.906**	**1.636**	**1.696**

TAB. 4.4 – Qualité des premières solutions.

mis en évidence dans le chapitre précédent sur les problèmes `ljb` que RINS et *guided dives* sont deux techniques qui permettent d'améliorer significativement la qualité des solutions entières obtenues. Sur ces modèles, *guided dives* obtient de meilleurs résultats que RINS, et RINS et *guided dives* utilisés de concert forment une stratégie hybride plus efficace que chacune des deux techniques utilisées séparément.

D'après les résultats de MIP 8, il semble que la résolution itérative ne permette pas d'améliorer significativement les performances. En moyenne, l'amélioration successive des coefficients M^i_{pq} n'est pas suffisante pour compenser le temps perdu à reprendre l'optimisation de zéro sur un nouveau modèle en perdant les informations accumulées dans l'arbre de recherche du modèle précédent, ce qui conduit à explorer plusieurs fois les mêmes régions de l'espace de recherche.

D'après les résultats de MIP 9, l'apport de l'heuristique MCORE est limité mais régulier. Sur quelques instances, cette heuristique trouve en explorant un seul voisinage des solutions améliorantes qui ne sont jamais trouvées ni par RINS ni par *guided dives*. En contrepartie, c'est une heuristique coûteuse qui ralentit souvent l'exploration arborescente. De plus, ne pouvant être invoquée que lorsqu'une nouvelle solution entière est découverte par ailleurs, il arrive sur certaines instances qu'elle ne soit pas appelée assez souvent pour modifier significativement les performances.

Pour finir, nous présentons dans le Tableau 4.5 les résultats de CP+LS et de la meilleure variante de programmation linéaire en nombres entiers (MIP 9), agrégés suivant l'ordre de grandeur de la taille des instances défini dans MaScLib. Nous avons inclus dans ce tableau toutes les instances de la série NCGS, en particulier les quatre plus grandes instances NCGS_75, NCGS_75a, NCGS_81 et NCGS_81a que nous n'avions pas considérées dans le Tableau 4.3. Ce tableau montre que la programmation linéaire en nombres entiers est meilleure que CP+LS sur les plus petites instances, mais moins robuste quand la taille des problèmes augmente, la limite se situant entre 75 et 100 opérations. Ceci est dû au fait que, pour les instances de grande taille, même la résolution de la relaxation linéaire est coûteuse et peu de nœuds peuvent être explorés dans l'arbre de *branch-and-cut*. Comme nous l'avons signalé à la Section 4.1.3, le nombre de variables binaires du modèle disjonctif croît environ avec le carré du nombre d'opérations en disjonction, alors que la limite de temps préconisée par MaScLib croît moins que linéairement avec la taille des instances.

Instances NCGS	1x	2x	3x	4x	5x	7x	8x
Nombre d'opérations (ordre de grandeur)	25	50	75	100	250	1000	2500
MIP 9	**1.000**	**1.000**	**1.030**	1.161	1.153	1.062	1.068
CP+LS	1.014	1.071	1.259	**1.159**	**1.061**	**1.013**	**1.005**

TAB. 4.5 – Qualité des solutions réalisables : résultats agrégés par taille.

4.3.4 Qualité des bornes inférieures

Pour évaluer la qualité des bornes inférieures obtenues par les différentes approches, nous donnons maintenant pour chaque algorithme d'une part le ratio entre la borne inférieure obtenue et la borne inférieure triviale égale au coût de mode, d'autre part le ratio entre la borne inférieure obtenue et le coût de la meilleure solution réalisable connue. La qualité de la borne inférieure croît donc avec le premier chiffre (qui est toujours supérieur à 1) et avec le deuxième chiffre (qui est toujours inférieur à 1). Les résultats agrégés sur l'ensemble des instances sont présentés dans le Tableau 4.6.

En résumé, les bornes inférieures produites par le modèle disjonctif sont de qualité médiocre, et calculer avec soin les coefficients des contraintes disjonctives ne permet pas de les améliorer. Ceci est particulièrement visible dans la première partie du Tableau 4.6 : la valeur de la relaxation linéaire au nœud racine pour le modèle naïf (MIP 1) et pour le modèle sophistiqué (MIP 3) est à peine supérieure à la borne inférieure triviale du coût de mode. Ces bornes inférieures sont tout de même améliorées d'abord grâce aux coupes ajoutées par CPLEX au nœud racine, puis grâce à l'exploration de l'arbre de *branch-and-cut*. Il est à noter que les stratégies mises en œuvre par exemple dans MIP 7 pour obtenir de meilleures solutions entières permettent aussi d'améliorer légèrement la qualité des bornes inférieures. En effet, si le *branch-and-cut* dispose d'une solution entière de relativement bonne qualité, alors il peut l'exploiter pour guider plus efficacement sa traversée de l'espace de recherche, ou au moins pour couper plus de branches. Les meilleurs coefficients calculés par la résolution itérative de MIP 8 ne permettent pas d'obtenir des bornes inférieures significativement meilleures.

Si l'on agrège les résultats critère par critère comme dans le Tableau 4.7, on s'aperçoit que les coupes ajoutées par CPLEX au nœud racine permettent d'améliorer notablement la qualité des bornes inférieures par rapport au coût de mode dans le cas de l'optimisation des coûts d'avance et retard (non-allocation autorisée ou non), ce qui ne se produit pas si l'on optimise seulement le coût de retard. Mais si l'on compare les bornes inférieures aux meilleures solutions connues, le *gap* est plus grand pour le critère d'avance et retard que pour le critère de retard seul.

Rappelons que CP+LS est un algorithme heuristique qui ne fournit pas de borne inférieure. Les bornes inférieures que nous fournissons, même si elles sont très éloignées des meilleures solutions entières connues, sont donc les seules disponibles à ce jour.

Algorithme	Comparaison avec le coût de mode	Comparaison avec la meilleure solution entière connue
Relaxation linéaire au nœud racine		
... MIP 1	1.054	0.544
... MIP 3	1.054	0.544
Au nœud racine après les coupes		
... MIP 1	1.305	0.674
... MIP 3	1.308	0.676
À la fin de la limite de temps		
... MIP 1	1.331	0.687
... MIP 3	1.331	0.688
... MIP 7	1.348	0.697
... MIP 8	1.350	0.697
... MIP 9	1.346	0.695

TAB. 4.6 – Qualité des bornes inférieures.

Algorithme	Critère d'optimisation				Moyenne générale
	ET	T	ET_UNP	T_UNP	
Comparaison avec le coût de mode					
Relaxation linéaire au nœud racine	1.065	1.049	1.051	1.049	1.054
Au nœud racine après les coupes	1.636	1.058	1.606	1.052	1.308
À la fin de la limite de temps	1.695	1.060	1.654	1.056	1.331
Comparaison avec la meilleure solution connue					
Relaxation linéaire au nœud racine	0.401	0.671	0.447	0.731	0.544
Au nœud racine après les coupes	0.615	0.677	0.683	0.732	0.676
À la fin de la limite de temps	0.637	0.680	0.701	0.736	0.688

TAB. 4.7 – Qualité des bornes inférieures de MIP 3 : résultats agrégés par critère.

4.4 Conclusion

Nous avons comparé dans ce chapitre plusieurs variantes de modélisation et plusieurs techniques de résolution pour le problème d'ordonnancement d'atelier avec coûts d'avance et retard. Nous avons montré que la programmation linéaire en nombres entiers permet d'obtenir des résultats tout à fait compétitifs sur le jeu de données MaScLib. Nos expériences ont confirmé que les techniques de résolution RINS et *guided dives* présentées au chapitre précédent sont efficaces sur ce problème, mais surtout qu'il est important de les combiner avec des stratégies spécifiques au modèle étudié telles qu'une bonne modélisation et une heuristique *ad hoc* de recherche de première solution pour obtenir une approche plus performante.

L'écart entre meilleure solution entière connue et borne inférieure est encore très important sur la plupart des instances de MaScLib. Il existe donc encore une grande marge d'amélioration. En particulier, il serait possible d'adapter les coupes introduites par Applegate et Cook [7] pour la minimisation de la durée totale de l'ordonnancement pour améliorer la qualité des bornes inférieures. Des essais numériques préliminaires sur quelques familles de coupes (*basic cuts*, *two-job cuts*, *triangle cuts*, *half cuts*, *late job cuts*) n'ont pas permis de mettre en évidence une amélioration significative, mais d'autres coupes spécifiques à la minimisation des coûts d'avance et retard peuvent sans doute être mises en œuvre.

Dans ce chapitre, nous avons introduit MCORE, une heuristique pour le *branch-and-cut* qui est spécifique aux modèles avec des coefficients de type *big-M*. MCORE appartient à la famille des algorithmes conceptuellement hybrides. Nos expériences numériques montrent que MCORE permet une petite amélioration des solutions entières. Il serait intéressant d'appliquer cette heuristique à d'autres modèles avec des coefficients de type *big-M* pour évaluer si la généralisation que nous en proposons est efficace.

Conclusion

Le premier objectif que nous nous étions fixé pour cette thèse était de développer des heuristiques de programmation linéaire en nombres entiers plus puissantes que les heuristiques de la littérature. Nous avons introduit l'heuristique RINS (*relaxation induced neighborhood search*) [43] qui est actuellement l'un des deux meilleurs algorithmes (avec la stratégie d'exploration arborescente *guided dives* [43]) pour trouver rapidement des solutions entières de qualité sur des modèles difficiles et quelconques de programmation linéaire en nombres entiers. RINS est implémenté dans ILOG CPLEX 9.0. Nous avons également proposé l'heuristique MCORE qui s'applique aux modèles avec des coefficients de type *big-M*. L'amélioration apportée par cette heuristique est moins significative et il n'est pas certain qu'elle puisse être généralisée telle quelle à d'autres problèmes que l'ordonnancement d'atelier avec coûts d'avance et retard, mais elle constitue une alternative prometteuse pour ces modèles particuliers. Nous avons enfin généralisé au *branch-and-price* l'idée des heuristiques pour le *branch-and-cut* [41] et nous avons appliqué avec succès ce schéma de coopération au problème de routage de véhicules avec fenêtres de temps. Ces quatre algorithmes sont quatre manières différentes d'intégrer les techniques de recherche locale à la programmation linéaire en nombres entiers.

Le deuxième objectif que nous nous étions fixé était de contribuer conceptuellement au domaine des algorithmes hybrides en général pour faciliter leur développement et l'extension de leur application. Nous avons dans un premier temps identifié deux schémas de coopération, la recherche multiple et la décomposition [40], qui permettent d'unifier et de classifier les algorithmes hybrides existants. Nous avons proposé en outre une catégorisation plus détaillée de la littérature des algorithmes hybrides où interviennent des techniques de recherche locale. Nous avons introduit dans un deuxième temps le paradigme des algorithmes conceptuellement hybrides [43] qui définit des algorithmes qui ne mettent en œuvre qu'une seule technique de résolution mais exploitent et transposent dans ce cadre les concepts de plusieurs techniques. Contrairement aux paradigmes existants, ce nouveau paradigme présente le triple avantage de ne pas augmenter la complexité logicielle de l'algorithme hybride par rapport à un algorithme pur, de limiter le risque de spécialisation de l'algorithme hybride à un problème particulier, et de faire facilement bénéficier l'algorithme hybride des progrès réalisés pour la technique d'optimisation sous-jacente. Nous avons montré avec RINS, *guided dives*

et MCORE que ce paradigme permet de développer des algorithmes simples, efficaces, robustes et génériques. Il englobe des algorithmes préexistants tels que le *local branching* [57] et certains algorithmes de *large neighborhood search*.

Notons enfin que nous avons essentiellement travaillé sur des techniques de résolution génériques, qui présentent l'intérêt de s'appliquer à n'importe quel modèle de programmation linéaire en nombres entiers sans nécessiter d'autre information que le modèle lui-même. Cependant, nous avons mis en évidence sur le problème d'ordonnancement d'atelier avec coûts d'avance et retard les limites de ces techniques de résolution génériques, en montrant d'une part qu'une bonne modélisation est complémentaire d'une bonne technique de résolution, d'autre part que les techniques de résolution spécifiques au problème particulier à résoudre sont complémentaires de techniques génériques.

Cette thèse ouvre plusieurs perspectives. Premièrement, le principe sur lequel est fondé RINS, qui consiste à définir un voisinage à l'aide de deux solutions de propriétés différentes (dont l'une au moins n'est pas réalisable pour le problème d'origine) est en fait très général. Nous avons utilisé pour RINS la solution entière courante et la solution de la relaxation continue, mais cette approche pourrait être généralisée à d'autres relaxations : relaxation continue du problème maître dans le cadre du *branch-and-price*, relaxation lagrangienne, relaxation semi-définie positive, ... Deux heuristiques de première solution pour la programmation linéaire en nombres entiers utilisent déjà ce principe en se servant de la solution de la relaxation continue d'une part, et d'un point fractionnaire à l'intérieur du polyèdre [55] ou d'une solution entière irréalisable [56] d'autre part. Ce principe pourrait être étendu au-delà de la programmation linéaire en nombres entiers, par exemple aux métaheuristiques à base de population de solutions, comme les algorithmes génétiques [104], le *scatter search* ou encore le *path relinking* [71], ou aux métaheuristiques qui construisent leurs voisinages à partir de l'historique des solutions comme certaines approches à base de grands voisinages [36].

Deuxièmement, notre travail sur les heuristiques pour le *branch-and-cut* et pour le *branch-and-price* a contribué à montrer que les heuristiques sont un moyen de résolution complémentaire de l'exploration arborescente, de la même façon que les coupes sont complémentaires du *branch-and-bound*. Après le *local branching*, RINS est un exemple d'une nouvelle génération d'heuristiques pour le *branch-and-cut* qui sont plus puissantes que les précédentes. Ce sujet de recherche a été relativement peu exploré, quatre problèmes en particulier nous semblent ouverts :

- L'efficacité et la robustesse : sur certains modèles, aucune heuristique existante ne réussit à améliorer la solution entière courante qui est pourtant loin de l'optimum. Il faudrait modifier la définition des voisinages de façon à ce qu'ils contiennent plus souvent des solutions améliorantes.
- La rapidité : RINS, le *local branching* et MCORE sont des heuristiques coûteuses qui ralentissent significativement l'exploration arborescente. Une manière d'accélérer ces heuristiques serait de construire des voisinages plus petits (ce qui

entre en conflit avec le point précédent) ou de sélectionner plus intelligemment les nœuds auxquels elles sont appelées.

– La recherche d'une première solution : RINS, le *local branching* et MCORE nécessitent une première solution entière pour être appelés, alors qu'il est parfois difficile d'en générer une. Les travaux récents de Fischetti *et al.* [56] constituent une première tentative dans cette direction.
– La spécialisation à des structures particulières : MCORE est une première tentative pour traiter spécifiquement les coefficients de type *big-M*. Des heuristiques particulières pourraient être mises en œuvre pour d'autres structures telles que les contraintes de type *generalized upper bound*, les différentes modélisations des fonctions linéaires par morceaux, les modèles de *set covering*, ...

Troisièmement, le paradigme des algorithmes conceptuellement hybrides semble à même de favoriser le développement, le déploiement et la généralisation des algorithmes hybrides. Par exemple, les deux idées suivantes nous semblent prometteuses :

– La transposition à la programmation linéaire en nombres entiers du concept de conflit en programmation par contraintes, comme le montrent les travaux en cours de Demassey [48] autour de *resolution search* [31], les travaux récents de Codato et Fischetti sur les coupes combinatoires de Benders [33] et les travaux de Chu et Xia sur les coupes entières de Benders [30].
– Des stratégies de branchement qui ne sont pas centrées sur l'impact du branchement sur la fonction objectif comme les traditionnels pseudo-coûts [99], mais qui prennent en compte à la place l'impact du branchement sur la géométrie de l'espace de recherche de manière analogue aux stratégies de branchement en programmation par contraintes. Les stratégies de branchement de Chinneck et Patel pour trouver plus rapidement une première solution entière [113] constituent un point de départ.

Annexe A

Sensibilité aux paramètres de RINS et du *local branching*

Comme nous l'avons mentionné au Chapitre 3, RINS et le *local branching* utilisent chacun deux paramètres dont il faut initialiser la valeur. Pour RINS, nous devons choisir la fréquence f à laquelle l'heuristique est appelée, et le nombre *nl* de nœuds à explorer dans le sous-modèle. Pour le *local branching*, nous devons choisir la valeur initiale r du rayon du voisinage, et le nombre *nl* de nœuds à explorer dans le sous-modèle. Nous rapportons ici les résultats obtenus avec différentes paramétrisations pour les mêmes expériences que celles de la Section 3.4.3 du Chapitre 3.

Le Tableau A.1 et les Figures A.1 à A.3 montrent que les performances de RINS sont très peu sensibles au changement de paramètres. Parmi les trois paramétrisations que nous avons expérimentées, aucune ne domine uniformément les autres. Par contre, le *local branching* est plus sensible au changement de paramètres, en particulier sur la classe des problèmes d'« écart important ». Il semble qu'explorer un voisinage plus petit ($r = 10$) ou arrêter plus tôt l'exploration du sous-modèle ($nl = 500$) permet d'améliorer légèrement les résultats. Ceci est à rapprocher de l'explication des mauvaises performances du *local branching* sur cette classe de problèmes que nous avons exposée en page 86.

Instance	RINS $f = 100$ $nl = 1000$	RINS $f = 50$ $nl = 1000$	RINS $f = 100$ $nl = 500$	LB $r = 20$ $nl = 1000$	LB $r = 10$ $nl = 1000$	LB $r = 20$ $nl = 500$
Problèmes de « faible écart » — une heure						
A1C1S1	1.006	**1.002**	1.010	1.010	1.011	1.013
A2C1S1	1.007	1.007	1.010	1.011	**1.000**	1.020
arki001	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
B2C1S1	1.041	1.044	**1.015**	1.079	1.079	1.079
biella1	**1.000**	1.002	**1.000**	1.001	1.001	**1.000**
nsrand_ipx	**1.000**	**1.000**	**1.000**	1.003	1.003	1.006
rail2586c	1.016	1.018	1.015	1.002	1.014	**1.000**
rail4284c	1.009	1.008	1.009	**1.000**	1.007	**1.000**
rail4872c	1.008	1.010	1.005	**1.003**	1.008	**1.003**
rococoB10-011000	**1.022**	1.037	**1.022**	1.057	1.034	1.053
rococoB10-011001	1.033	1.047	1.042	1.124	**1.025**	1.060
rococoB11-010000	1.061	1.079	**1.055**	1.092	1.118	1.094
rococoC10-001000	1.001	**1.000**	1.001	1.003	1.001	1.002
roll3000	1.005	**1.001**	1.005	**1.001**	1.003	1.003
seymour	1.002	**1.000**	1.002	1.009	1.009	1.009
sp97ar	**1.001**	1.002	**1.001**	1.014	1.012	1.010
sp97ic	**1.002**	1.004	**1.002**	1.004	1.012	1.004
sp98ar	**1.002**	**1.002**	**1.002**	1.004	**1.002**	1.007
sp98ic	**1.003**	**1.003**	**1.003**	1.012	1.004	1.009
tr12-30	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
UMTS	**1.000**	**1.000**	**1.000**	1.002	1.001	**1.000**
Problèmes d'« écart intermédiaire » — une heure						
B1C1S1	1.011	**1.008**	1.012	1.045	1.020	1.028
glass4	1.096	1.109	**1.014**	1.096	1.130	1.130
ljb2	**1.000**	1.033	**1.000**	1.225	1.116	1.200
net12	**1.000**	**1.000**	**1.000**	1.192	1.192	1.192
rococoB11-110001	1.121	1.141	**1.085**	1.260	1.254	1.229
rococoB12-111111	1.029	**1.015**	1.028	1.085	1.057	1.085
rococoC10-100001	**1.095**	1.126	**1.095**	1.314	1.214	1.302
rococoC11-011100	1.081	**1.042**	1.046	1.278	1.081	1.277
rococoC11-010100	**1.055**	**1.055**	**1.055**	1.180	1.330	1.180
rococoC12-100000	1.096	1.029	**1.024**	1.172	1.110	1.127
rococoC12-111100	**1.025**	1.057	1.038	1.135	1.125	1.147
swath	1.048	**1.042**	1.048	1.090	1.154	1.126
Problèmes d'« écart important » — deux heures						
ljb7	1.061	**1.000**	1.041	1.583	1.580	1.983
ljb9	1.581	1.718	**1.548**	2.337	1.995	2.254
ljb10	1.212	**1.097**	1.253	2.471	1.693	1.416
ljb12	**1.512**	1.692	1.717	3.164	3.083	4.795

TAB. A.1 – Ratio solution obtenue / meilleure solution connue.

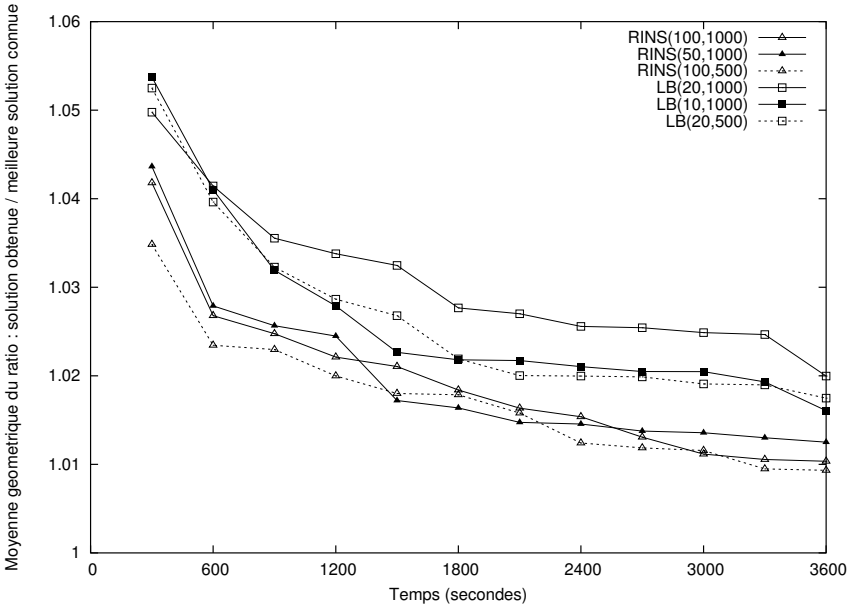

FIG. A.1 – Problèmes de « faible écart ».

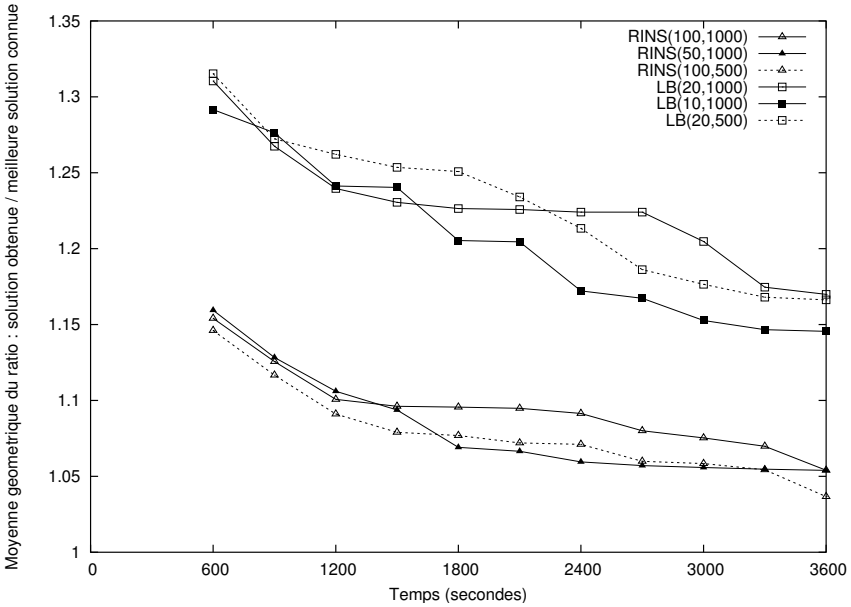

FIG. A.2 – Problèmes d’« écart intermédiaire ».

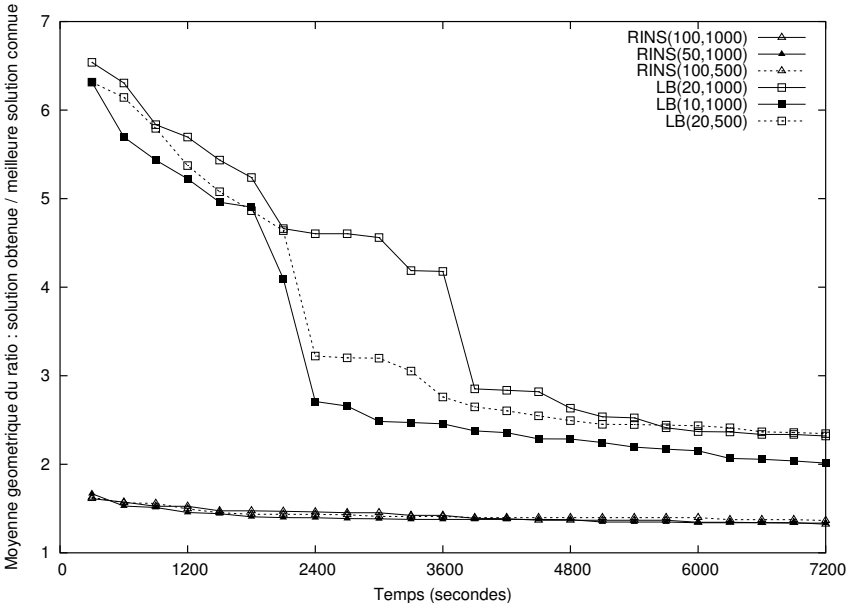

FIG. A.3 – Problèmes d'« écart important ».

Bibliographie

[1] Emile H.L. AARTS et Jan Karel LENSTRA. *Local Search in Combinatorial Optimization*. Wiley, 1997.

[2] Ronny ABOUDI et Kurt JÖRNSTEN. « Tabu search for general zero-one integer programs using the pivot and complement heuristic ». *ORSA Journal on Computing*, 6(1) : 82–93, 1994.

[3] D. ABRAMSON, H. DANG, et M. KRISHNAMOORTHY. « A comparison of two methods for solving 0-1 integer programs using a general purpose simulated annealing algorithm ». *Annals of Operations Research*, 63 : 129–150, 1996.

[4] Joseph ADAMS, Egon BALAS, et Daniel ZAWACK. « The shifting bottleneck procedure for job-shop scheduling ». *Management Science*, 34(3) : 391–401, 1988.

[5] Ravindra K. AHUJA, Özlem ERGUN, James B. ORLIN, et Abraham P. PUNNEN. « A survey of very large-scale neighborhood search techniques ». *Discrete Applied Mathematics*, 123 : 75–102, 2002.

[6] R. ANBIL, E. GELMAN, B. PATTY, et R. TANGA. « Recent advances in crew-pairing optimization at American Airlines ». *Interfaces*, 21 : 62–74, 1991.

[7] David APPLEGATE et William COOK. « A computational study of the job-shop scheduling problem ». *ORSA Journal on Computing*, 3(2) : 149–156, 1991.

[8] Egon BALAS. « The intersection cut — a new cutting plane for integer programming ». *Operations research*, 19(1) : 19–39, 1971.

[9] Egon BALAS, Sebastián CERIA, Milind DAWANDE, François MARGOT, et Gábor PATAKI. « OCTANE : a new heuristic for pure 0-1 programs ». *Operations Research*, 49(2) : 207–225, 2001.

[10] Egon BALAS et Clarence H. MARTIN. « Pivot and complement — a heuristic for 0-1 programming ». *Management Science*, 26(1) : 86–96, 1980.

[11] Philippe BAPTISTE et Francis SOURD. « Job-shop scheduling to minimize earliness and tardiness ». In *Proceedings of the Ninth International Workshop on Project Management and Scheduling (PMS'04)*, pages 246–249, 2004.

[12] Cynthia BARNHART, Ellis L. JOHNSON, George L. NEMHAUSER, Martin W.P. SAVELSBERGH, et Pamela H. VANCE. « Branch-and-price : column generation for solving huge integer programs ». *Operations Research*, 46 : 316–329, 1998.

[13] J. Christopher BECK et Philippe REFALO. « A hybrid approach to scheduling with earliness and tardiness costs ». In *Proceedings of the Third International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'01)*, pages 175–188, 2001.

[14] J. Christopher BECK et Philippe REFALO. « Combining local search and linear programming to solve earliness/tardiness scheduling problems ». In *Proceedings of the Fourth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'02)*, pages 221–235, 2002.

[15] J. Christopher BECK et Philippe REFALO. « A hybrid approach to scheduling with earliness and tardiness costs ». *Annals of Operations Research*, 118(1-4) : 49–71, 2003.

[16] Russel BENT et Pascal VAN HENTENRYCK. « A two-stage hybrid local search for the vehicle routing problem with time windows ». Technical Report CS-01-06 (à paraître dans *Transportation Science*, Brown University, 2001.

[17] Robert E. BIXBY, Sebastián CERIA, Cassandra M. MCZEAL, et Martin W. P. SAVELSBERGH. « An updated mixed integer programming library : MIPLIB 3.0 ». *Optima*, 58 : 12–15, 1998.

[18] Robert E. BIXBY, Mary FENELON, Zonghao GU, Edward ROTHBERG, et Roland WUNDERLING. MIP : theory and practice — closing the gap. In *System Modelling and Optimization : Methods, Theory, and Applications*, pages 19–49. Kluwer Academic Publishers, 2000.

[19] K. BUDENBENDER, T. GRUNERT, et H.-J. SEBASTIAN. « A hybrid tabu search/branch-and-bound algorithm for the direct flight network design problem ». *Transportation Science*, 34(4) : 364–380, 2000.

[20] Jacques CARLIER. « The one-machine sequencing problem ». *European Journal of Operational Research*, 11(1) : 42–47, 1982.

[21] Jacques CARLIER et Éric PINSON. « An algorithm for solving the job-shop problem ». *Management Science*, 35(2) : 164–176, 1989.

[22] Yves CASEAU, François LABURTHE, Claude LE PAPE, et Benoît ROTTEMBOURG. « Combining local and global search in a constraint programming environment ». *Knowledge Engineering Review*, 16(1) : 41–68, 2001.

[23] Yves CASEAU et François LABURTHE. « Disjunctive scheduling with tasks intervals ». Technical Report LIENS-95-25, École Normale Supérieure, Département de mathématiques et informatique, 1995.

[24] Yves CASEAU et François LABURTHE. « Effective forget-and-extend heuristics for scheduling problems ». In *Proceedings of the First International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'99)*, 1999.

[25] C.C.B. CAVALCANTE, C. Caravlho de SOUZA, Martin W.P. SAVELSBERGH, Y. WANG, et L.A. WOLSEY. « Scheduling projects with labor constraints ». *Discrete Applied Mathematics*, 112 : 27–52, 2001.

[26] Alain CHABRIER. « Vehicle routing problem with elementary shortest path based column generation ». *À paraître dans Computers and Operations Research*, 2003.

[27] Alain CHABRIER, Emilie DANNA, et Claude LE PAPE. « Coopération entre génération de colonnes avec tournées sans cycle et recherche locale appliquée au routage de véhicules ». In *Huitièmes Journées Nationales sur la résolution de Problèmes NP-Complets (JNPC'2002)*, pages 83–97, 2002.

[28] Alain CHABRIER, Emilie DANNA, Claude LE PAPE, et Laurent PERRON. « Solving a network design problem ». *Annals of Operations Research (à paraître)*.

[29] Haoxu CHEN, Chengbin CHU, et Jean-Marie PROTH. « An improvement of the Lagrangean relaxation approach for job shop scheduling : a dynamic programming method ». *IEEE Transactions on Robotics and Automation*, 14 : 786–795, 1998.

[30] Yingyi CHU et Quanshi XIA. « Generating Benders cuts for a general class of integer programming problems ». In *Proceedings of the First International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'04)*, pages 127–141, 2004.

[31] Vasek CHVÁTAL. « Resolution search ». *Discrete Applied Mathematics*, 73 : 81–99, 1997.

[32] G. CLARKE et J.W. WRIGHT. « Scheduling of vehicles from a central depot to a number of delivery points ». *Operations Research*, 12(4) : 568–581, 1964.

[33] Gianni CODATO et Matteo FISCHETTI. « Combinatorial Benders's cuts ». *In proceedings of IPCO X*, 2004.

[34] David T. CONNOLLY. « General purpose simulated annealing ». *Journal of the Operational Research Society*, 43 : 495–505, 1992.

[35] William COOK et Jennifer L. RICH. « A parallel cutting-plane algorithm for the vehicle routing problem with time windows ». Technical Report TR99-04, Department of Computational and Applied Mathematics, Rice University, 1999.

[36] William COOK et Paul SEYMOUR. « Tour merging via branch-decomposition ». *INFORMS Journal on Computing*, 15(3) : 233–248, 2003.

[37] Jean-François CORDEAU, Guy DESAULNIERS, Jacques DESROSIERS, Marius M. SOLOMON, et François SOUMIS. The VRP with time windows. In Paolo

TOTH et Daniele VIGO, editors, *The Vehicle Routing Problem*, pages 157–193. SIAM Monographs on Discrete Mathematics and Applications, 2002.

[38] Jean-François CORDEAU, Gilbert LAPORTE, et Anne MERCIER. « A unified tabu search heuristic for vehicle routing problems with time windows ». *Journal of the Operational Research Society*, 52 : 928–936, 2001.

[39] Emilie DANNA. « Improving and extending the disjunctive MIP model for the earliness-tardiness job-shop scheduling problem ». In *Proceedings of the Ninth International Workshop on Project Management and Scheduling (PMS'04)*, pages 250–253, 2004.

[40] Emilie DANNA et Claude LE PAPE. Two generic schemes for efficient and robust cooperative algorithms. In Michela MILANO, editor, *Constraint and integer programming*, pages 33–57. Kluwer Academic Publishers, 2004.

[41] Emilie DANNA et Claude Le PAPE. Branch-and-price heuristics : A case study on the vehicle routing problem with time windows. In Guy DESAULNIERS, Jacques DESROSIERS, et Marius M. SOLOMON, editors, *Book on column generation for the* 25^{th} *GERAD anniversary*. Kluwer Academic Publishers. En révision.

[42] Emilie DANNA et Laurent PERRON. « Structured vs. unstructured large neighborhood search : a case study on job-shop scheduling problems with earliness and tardiness costs ». In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP'03)*, pages 817–821, 2003.

[43] Emilie DANNA, Edward ROTHBERG, et Claude LE PAPE. « Exploring relaxation induced neighborhoods to improve MIP solutions ». À paraître dans Mathematical Programming.

[44] Emilie DANNA, Edward ROTHBERG, et Claude Le PAPE. « Integrating mixed integer programming and local search : A case study on job-shop scheduling problems ». In *Proceedings of the Fifth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'03)*, pages 65–79, 2003.

[45] Stéphane DAUZÈRE-PÉRÈS et Jean B. LASSERRE. « A new mixed-integer formulation of the flow-shop sequencing problem ». In *Proceedings of the Second Workshop on Models and Algorithms for Planning and Scheduling Problems*, 1995.

[46] Bruno DE BACKER, Vincent FURNON, Paul SHAW, Philip KILBY, et Patrick PROSSER. « Solving vehicle routing problems using constraint programming and metaheuristics ». *Journal of Heuristics*, 6 : 501–523, 2000.

[47] Thierry DEFAIX. « Challenge ROADEF 2001 ». `http://www.prism.uvsq.fr/~vdc/ROADEF/CHALLENGES/2001/`, 2000.

[48] Sophie DEMASSEY. « *Méthodes hybrides de programmation par contraintes et programmation linéaire pour le problème d'ordonnancement de projet à contraintes de ressources* ». PhD thesis, Université d'Avignon et des Pays du Vaucluse, 2003.

[49] Guy DESAULNIERS, Jacques DESROSIERS, et Marius M. SOLOMON. Accelerating strategies for column generation methods in vehicle routing and crew scheduling problems. In C.C. RIBEIRO et P. HANSEN, editors, *Essays and Surveys in Metaheuristics*, pages 309–324. Kluwer, 2002.

[50] Martin DESROCHERS. « *La fabrication d'horaires de travail pour les conducteurs d'autobus par une méthode de génération de colonnes* ». PhD thesis, Université de Montréal, Canada, 1986.

[51] Martin DESROCHERS, Jacques DESROSIERS, et Marius M. SOLOMON. « A new optimization algorithm for the vehicle routing problem with time windows ». *Operations Research*, 40 : 342–354, 1992.

[52] Irina DUMITRESCU et Thomas STÜZLE. « A survey of methods that combine local search and exact algorithms ». Soumis à European Journal of Operations Research, 2003.

[53] Martin E. DYER et Laurence A. WOLSEY. « Formulating the single machine sequencing problem with release dates as a mixed integer program ». *Discrete Applied Mathematics*, 24 : 255–270, 1990.

[54] Kelly EASTON, George NEMHAUSER, et Michael TRICK. CP based branch-and-price. In Michela MILANO, editor, *Constraint and Integer Programming*, pages 207–231. Kluwer Academic Publishers, 2004.

[55] Bruce H. FAALAND et Frederick S. HILLIER. « Interior path methods for heuristic integer programming procedures ». *Operations Research*, 27(6) : 1069–1087, 1979.

[56] Matteo FISCHETTI, Fred GLOVER, et Andrea LODI. « The feasibility pump ». Communication privée, 2003.

[57] Matteo FISCHETTI et Andrea LODI. « Local branching ». *Mathematical Programming, Series B*, 98 : 23–47, 2003.

[58] Filippo FOCACCI, Andrea LODI, et Michela MILANO. Exploiting relaxations in CP. In Michela MILANO, editor, *Constraint and Integer Programming*, pages 137–167. Kluwer Academic Publishers, 2004.

[59] Filippo FOCACCI et Paul SHAW. « Pruning sub-optimal search branches using local search ». In *Proceedings of the Fourth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'02)*, pages 181–189, 2002.

[60] Mark S. FOX et Stephen F. SMITH. « ISIS : a knowledge-based system for factory scheduling ». *Expert Systems*, 1(1) : 25–49, 1984.

[61] Alan P. FRENCH, Andrew P. ROBINSON, et John M. WILSON. « Using a hybrid genetic-algorithm/branch and bound approach to solve feasibility and optimization integer programming problems ». *Journal of Heuristics*, 7 : 551–564, 2001.

[62] Birger FUNKE. « *Effiziente Lokale Suche für vehicle routing und scheduling Probleme mit Ressourcenbeschränkungen* ». PhD thesis, Fakultät für Wirtschaftswissenschaften, RWTH Aachen, Germany, 2003.

[63] Virginie GABREL, Arnaud KNIPPEL, et Michel MINOUX. « A comparison of heuristics for the discrete cost multicommodity network optimization problem ». *Journal of Heuristics*, 9 : 429–445, 2003.

[64] Luca Maria GAMBARDELLA, Éric D. TAILLARD, et Giovanni AGAZZI. MACS-VRPTW : a multiple ant colony system for vehicle routing problems with time windows. In David CORNE, Marco DORIGO, et Fred GLOVER, editors, *New ideas in optimization*, pages 63–76. McGraw-Hill, London, 1999.

[65] M.R. GAREY, R.E. TARJAN, et G.T. WILFONG. « One-processor scheduling with symmetric earliness and tardiness penalties ». *Mathematics of Operations Research*, 13 : 330–348, 1988.

[66] Fred GLOVER. « Scatter search and star paths : beyond the genetic metaphor ». *OR Spektrum*, 17 : 125–137, 1995.

[67] Fred GLOVER et Arne LØKKETANGEN. « Solving zero-one mixed integer programming problems using tabu search ». *European Journal of Operational Research*, 106 : 624–658, 1997.

[68] Fred GLOVER et Manuel LAGUNA. « General purpose heuristics for integer programming – part I ». *Journal of Heuristics*, 2(4) : 343–358, 1997.

[69] Fred GLOVER et Manuel LAGUNA. « General purpose heuristics for integer programming – part II ». *Journal of Heuristics*, 3(2) : 161–179, 1997.

[70] Fred GLOVER et Manuel LAGUNA. *Tabu search*. Kluwer Academic Publishers, 1997.

[71] Fred GLOVER, Manuel LAGUNA, et Rafael MARTI. « Fundamentals of scatter search and path relinking ». *Control and Cybernetics*, 29(3) : 653–684, 2000.

[72] Fred GLOVER, Arne LØKKETANGEN, et David L. WOODRUFF. Scatter search to generate diverse MIP solutions. In M. LAGUNA et J.L. GONZÁLEZ-VELARDE, editors, *OR Computing Tools for Modeling, Optimization and Simulation : Interfaces in Computer Science and Operations Research*, pages 299–317. Kluwer Academic Publishers, 2000.

[73] Carla P. GOMES et Bart SELMAN. « Algorithm portfolios ». *Artificial Intelligence*, 126(1-2) : 43–62, 2001.

[74] Carla P. GOMES, Bart SELMAN, et Henri KAUTZ. « Boosting combinatorial search through randomization ». In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, pages 431–437, 1998.

[75] Cyrille GUÉGUEN, Pierre DEJAX, Moshe DROR, Dominique FEILLET, et Michel GENDREAU. « An exact algorithm for the elementary shortest path problem with resource constraints : application to some vehicle routing problems ». Technical Report CRT-2000-15 (à paraître dans *Networks*), Center for Research on Transportation, Montréal, 2000.

[76] Pierre HANSEN et Nenad MLADENOVIĆ. « Variable neighborhood search ». *Computers and Operations Research*, 24 : 1097–1100, 1997.

[77] William D. HARVEY et Matthew L. GINSBERG. « Limited Discrepancy Search ». In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI'95) ; Vol. 1*, pages 607–615, 1995.

[78] Yann HENDEL et Francis SOURD. « Calcul de voisinages pour l'ordonnancement avec critères irréguliers ». In *Conférence Francophone de Modélisation et Simulation (MOSIM'03)*, pages 21–25, 2003.

[79] Yann HENDEL et Francis SOURD. « Efficient neighborhood search for just-in-time scheduling problems ». Soumis à European Journal of Operational Research, 2003.

[80] Frederick S. HILLIER. « Efficient heuristic procedures for integer linear programming with an interior ». *Operations Research*, 17(4) : 600–637, 1969.

[81] K. HOFFMAN et M. PADBERG. « Improving representations of zero-one linear programs for branch-and-cut ». *ORSA Journal of Computing*, 3 : 121–134, 1991.

[82] Jörg HOMBERGER et Hermann GEHRING. « Two evolutionary metaheuristics for the vehicle routing problem with time windows ». *INFOR*, 37 : 297–318, 1999.

[83] Han HOOGEVEEN. « Multicriteria scheduling ». Technical Report ALCOM-FT-TR-03-190, Utrecht University, 2003. À paraître dans European Journal of Operations Research.

[84] J.A. HOOGEVEEN et S.L. van de VELDE. « A branch-and-bound algorithm for single-machine earliness-tardiness scheduling with idle time ». *INFORMS Journal on Computing*, 8 : 402–412, 1996.

[85] Toshihide IBARAKI, Tateaki OHASHI, et Hisashi MINE. « A heuristic algorithm for mixed-integer programming problems ». *Mathematical Programming Study*, 2 : 115–136, 1974.

[86] ILOG, S.A.. *ILOG DISPATCHER 3.3 User's Manual*. 2002.

[87] Stephan IRNICH. « The shortest path problem with k-cycle elimination ($k \geq 3$) : improving a branch and price algorithm for the VRPTW ». Technical Report, Lehr- und Forschungsgebiet Unternehmensforshung (Operations Research) Rheinish-Westfälishe Technische Hochshule (RWTH) Aachen, Germany, 2001.

[88] Stephan IRNICH et Daniel VILLENEUVE. « The shortest path problem with resource constraints and k-cycle elimination for $k \geq 3$ ». Technical Report G-2003-55, GERAD, 2003.

[89] Ulrich JUNKER et Wim NUIJTEN. « Preference-based search for minimizing changes in rescheduling problems ». In *Proceedings of the IJCAI-99 Workshop on Scheduling and Planning meet Real-time Monitoring in a Dynamic and Uncertain World*, pages 39–45, 1999.

[90] Brian KALLEHAUGE, Jesper LARSEN, et Oli B.G. MADSEN. « Lagrangean duality and non-differentiable optimization applied on routing with time windows — experimental results ». Technical Report IMM-TR-2001-9, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark, 2001.

[91] Philip KILBY, Patrick PROSSER, et Paul SHAW. « A comparison of traditional and constraint-based heuristic methods on vehicle routing problems with side constraints ». *Constraints*, 5(4) : 389–414, 2000.

[92] Arne LØKKETANGEN et Fred GLOVER. Probabilistic move selection in tabu search for zero-one mixed integer programming problems. In I.H. OSMAN et J.P. KELLY, editors, *Metaheuristics : Theory and Applications*, pages 555–570. Kluwer Academic Publishers, 1996.

[93] Arne LØKKETANGEN et Fred GLOVER. Candidate list and exploration strategies for solving 0/1 MIP problems using a pivot neighborhood. In S. VOSS, S. MARTELLO, I.H. OSMAN, et C. ROUCAIROL, editors, *Metaheuristics : advances and trends in local search paradigms for optimization*, pages 141–155. Kluwer Academic Publishers, 1998.

[94] Niklas KOHL, Jacques DESROSIERS, Oli B.G. MADSEN, Marius M. SOLOMON, et François SOUMIS. « 2-Path cuts for the vehicle routing problem with time windows ». *Transportation Science*, 1(13) : 101–116, 1999.

[95] A.W.J. KOLEN, A.H.G. RINNOOY KAN, et H.W.J.M. TRIENEKENS. « Vehicle routing with time windows ». *Operations Research*, 35 : 266–273, 1987.

[96] Jesper LARSEN. « *Parallelization of the vehicle routing problem with time windows* ». PhD thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 1999.

[97] Jean B. LASSERRE et Maurice QUEYRANNE. « Generic scheduling polyhedra and a new mixed integer formulation for single machine scheduling ». In *Proceedings of the Second Integer Programming and Combinatorial Optimization (IPCO) Conference*, 1992.

[98] Claude LE PAPE et Philippe BAPTISTE. « Heuristic control of a constraint-based algorithm for the preemptive job-shop scheduling problem ». *Journal of Heuristics*, 5(3) : 305–325, 1999.

[99] J.T. LINDEROTH et Martin W.P. SAVELSBERGH. « A computational study of search strategies for mixed integer programming ». *INFORMS Journal on Computing*, 11 : 173–187, 1999.

[100] Arne LØKKETANGEN et David L. WOODRUFF. « Integrating pivot based search with branch and bound for binary MIP's ». *Control and Cybernetics, Special issue on Tabu Search*, 29(3) : 741–760, 2001.

[101] H.R. LOURENÇO, O. MARTIN, et T. STÜLZE. Iterated local search. In G. KOCHENBERGER et F. GLOVER, editors, *Handbook of metaheuristics*, pages 321–353. Kluwer Academics Publishers, 2002.

[102] Thierry MAUTOR et Philippe MICHELON. « Mimausa : an application of referent domain optimization ». Technical Report LIA-260, Laboratoire d'Informatique d'Avignon, 2001.

[103] Michela MILANO et Willem J. van HOEVE. « Reduced cost-based ranking for generating promising subproblems ». In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP'02)*, pages 1–16, 2002.

[104] Melanie MITCHELL. *An introduction to genetic algorithms*. MIT Press, 1996.

[105] Halim M'SILTI, Pierre TOLLA, et Arnaud SCHAAL. « Comparison of simulated annealing and genetic algorithm with a hybrid method for general integer problems ». Technical Report Cahier n°147, Laboratoire d'analyse et modélisation de systèmes pour l'aide à la décision (LAMSADE), 1997.

[106] B. NADEL. Tree search and arc consistency in constraint satisfaction algorithms. In L. KANAL et V. KUMAR, editors, *Search in artificial intelligence*, pages 287–342. Springer Verlag, 1988.

[107] Mikhail NEDIAK et Jonathan ECKSTEIN. « Pivot, cut and dive : a heuristic for 0-1 mixed integer programming ». Technical Report RRR 53-2001, Rutgers Center for Operations Research, 2001.

[108] George NEMHAUSER et Lawrence WOLSEY. *Integer and Combinatorial Optimization*. John Wiley and Sons, New York, 1988.

[109] Wim NUIJTEN, Thomas BOUSONVILLE, Filippo FOCACCI, Daniel GODARD, et Claude LE PAPE. « Towards an industrial manufacturing scheduling problem and test bed ». In *Proceedings of the Ninth International Workshop on Project Management and Scheduling (PMS'04)*, pages 162–165, 2004.

[110] Wim NUIJTEN et Claude LE PAPE. « Constraint-based job-shop scheduling with ILOG Scheduler ». *Journal of Heuristics*, 3(4) : 271–286, 1998.

[111] Mireille PALPANT, Christian ARTIGUES, et Philippe MICHELON. « A heuristic for solving the frequency assignment problem ». In *Proceedings of the XI Latin-Iberian American Congress of Operations Research (CLAIO)*, 2002.

[112] Mireille PALPANT, Christian ARTIGUES, et Philippe MICHELON. « Solving the resource-constrained project scheduling problem by integrating exact resolution and local search ». In *Proceedings of the Eighth International Workshop on Project Management and Scheduling (PMS'02)*, pages 289–292, 2002.

[113] Japat PATEL et John W. CHINNECK. « Faster MIP Solutions Through Better Variable Ordering ». In *18th International Symposium on Mathematical Programming (ISMP'03)*, 2003.

[114] Laurent PERRON. « Fast restart policies and large neighborhood search ». In *Proceedings of the Fifth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'03)*, pages 246–260, 2003.

[115] Gilles PESANT et Michel GENDREAU. « A view of local search in constraint programming ». In *Proceddings of the Second International Conference on Principles and Practice of Constraint Programming (CP'96)*, pages 353–366, 1996.

[116] Stefan RØPKE. « A general heuristic for vehicle routing problems ». In *International Workshop on Vehicle Routing and Multi-modal Transportation*, page 46, 2003.

[117] Stefan RØPKE. « A local search heuristic for the pickup and delivery problem with time windows ». Technical Report, DIKU, University of Copenhagen, 2003.

[118] Agnès PLATEAU, Dominique TACHAT, et Pierre TOLLA. « A hybrid search combining interior point methods and metaheuristics for 0-1 programming ». *International Transactions in Operational Research*, 9 : 731–746, 2002.

[119] Maurice QUEYRANNE et Andreas S. SCHULZ. « Polyhedral approaches to machine scheduling problems ». Technical Report 208/1994, Department of Mathematics, Technische Universität Berlin, Germany, 1994.

[120] P. RAGHAVAN et C.D. THOMPSON. « Randomized rounding : a technique for provably good algorithms and algorithmic proofs ». *Combinatorica*, 7(4) : 365–374, 1987.

[121] Jean-Charles RÉGIN. « A filtering algorithm for constraints of difference in CSPs ». In *Proceedings of the National Conference on Artificial Intelligence (AAAI-98)*, pages 362–367, 1994.

[122] Jean-Charles RÉGIN. Global constraints and filtering algorithms. In Michela MILANO, editor, *Constraint and Integer Programming*, pages 89–135. Kluwer Academic Publishers, 2004.

[123] Éric D. TAILLARD et S. VOSS. POPMUSIC : Partial optimization metaheuristic under special intensification conditions. In C. RIBEIRO et P. HANSEN, editors, *Essays and surveys in metaheuristics*, pages 613–629. Kluwer, 2001.

[124] Yves ROCHAT et Éric D. TAILLARD. « Probabilistic diversification and intensification in local search for vehicle routing ». *Journal of Heuristics*, 1 : 147–167, 1995.

[125] Louis-Martin ROUSSEAU, Michel GENDREAU, et Gilles PESANT. « Solving small VRPTWs with constraint programming based column generation ». In *Proceedings of the Fourth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'02)*, pages 333–344, 2002.

[126] Louis-Martin ROUSSEAU, Michel GENDREAU, et Gilles PESANT. « Using constraint-based operators to solve the vehicle routing problem with time windows ». *Journal of Heuristics*, 8(1) : 43–58, 2002.

[127] Norman SADEH et Mark S. FOX. « Variable and value ordering heuristics for activity-based job-shop scheduling ». In *Proceedings of the Fourth International Conference on Expert Systems in Production and Operations Management*, 1990.

[128] Hani El SAKKOUT et Mark WALLACE. « Probe backtrack search for minimal perturbation in dynamic scheduling ». *Constraints*, 5(4) : 359–388, 2000.

[129] Martin W.P. SAVELSBERGH. « Preprocessing and probing for mixed integer programming problems ». *ORSA Journal on Computing*, 6 : 445–454, 1994.

[130] Martin W.P. SAVELSBERGH et Marc SOL. « DRIVE : Dynamic Routing of Independent VEhicles ». *Operations Research*, 46(4) : 474–490, 1998.

[131] Arnaud SCHAAL, Halim M'SILTI, et Pierre TOLLA. « Une approche hybride de résolution de problèmes linéaires généraux en nombres entiers ». Technical Report Cahier n°145, Laboratoire d'analyse et modélisation de systèmes pour l'aide à la décision (LAMSADE), 1997.

[132] Robin SCHILHAM. « *Commonalities in local search* ». PhD thesis, Technische Universiteit Eindhoven, 2001.

[133] Alexander SCHRIJVER. *Theory of linear and integer programming*. Wiley, 1987.

[134] Bart SELMAN, Henry A. KAUTZ, et Bram COHEN. « Noise strategies for improving local search ». In *Proceedings of the Twelfth National Conference on Artificial intelligence (AAAI-94)*, pages 337–343, 1994.

[135] Paul SHAW. « Using constraint programming and local search methods to solve vehicle routing problems ». In *Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming (CP'98)*, pages 417–431, 1998.

[136] Marius M. SOLOMON. « Algorithms for the vehicle routing and scheduling problem with time window constraints ». *Operations Research*, 35 : 254–265, 1987.

[137] Francis SOURD. « Scheduling a sequence of tasks with general completion costs ». Technical Report 2002/013, LIP6, 2002.

[138] Francis SOURD et S. KEDAD-SIDHOUM. « The one machine scheduling with earliness and tardiness penalties ». *Journal of Scheduling*, 6(6) : 533–549, 2003.

[139] Éric D. TAILLARD. « Parallel iterative search methods for vehicle routing problems ». *Networks*, 23 : 661–676, 1993.

[140] Éric D. TAILLARD. « A heuristic column generation method for the heterogeneous VRP ». *Operations Research — Recherche Opérationnelle*, 33(1) : 1–14, 1999.

[141] Éric D. TAILLARD. « Heuristic methods for large centroid clustering problems ». *Journal of Heuristics*, 9(1) : 51–73, 2003.

[142] Éric D. TAILLARD, Philippe BADEAU, Michel GENDREAU, François GUERTIN, et Jean-Yves POTVIN. « A tabu search heuristic for the vehicle routing problem with soft time windows ». *Transportation Science*, 31 : 170–186, 1997.

[143] Leonardo Bedoya VALENCIA et Ghaith RABADI. « A multiagents approach for the jobshop shceduling problem with earliness and tardiness ». In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, pages 1217–1222, 2003.

[144] J.M. van den AKKER, C.A.J. HURKENS, et Martin W.P. SAVELSBERGH. « Time-indexed formulations for machine scheduling problems : column generation ». *INFORMS Journal on Computing*, 12, 1998.

[145] Pascal VAN HENTENRYCK et Thierry LE PROVOST. « Incremental search in constraint logic programming ». *New Generation Computing*, 9(3) : 257–275, 1991.

[146] Willem J. van HOEVE. « A hybrid constraint programming and semidefinite programming approach for the stable set problem ». In *Proceedings of the Fifth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'03)*, pages 3–16, 2003.

[147] Peter J.M. VAN LAARHOVEN, Emile H.L. AARTS, et Jan Karel LENSTRA. « Job shop scheduling by simulated annealing ». *Operations Research*, 40(1), 1992.

[148] Peter J.M. VAN LAARHOVEN et Emile H.L. AARTS. *Simulated Annealing : Theory and Practice*. Kluwer Academic Publishers, 1987.

[149] Mathieu VAN VYVE. « *A solution approach of production planning problems based on compact formulations for single-item lot-sizing models* ». PhD thesis, Université catholique de Louvain-la-Neuve, 2003.

[150] Mario VANHOUCKE, Erik DEMEULEMEESTER, et Willy HERROELEN. « An exact procedure for the resource-constrained weighted earliness-tardiness project scheduling problem ». *Annals of Operations Research*, 102 : 179–196, 2001.

[151] Michel VASQUEZ. « Arc-consistency and tabu search for the frequency assignement problem with polarization ». In *Proceedings of the Fourth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'02)*, pages 359–372, 2002.

[152] Michel VASQUEZ et Jin-Kao HAO. « A hybrid approach for the 0-1 multidimensional knapsack problem ». In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'2001)*, pages 328–333, 2001.

[153] Manuel VÁZQUEZ et L. Darrell WHITLEY. « A comparison of genetic algorithms for the dynamic job shop scheduling problem ». In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 1011–1018, 2000.

[154] Gérard VERFAILLIE, Michel LEMAITRE, et Thomas SCHIEX. « Russian doll search for solving constraint optimization problems ». In *Procedings of the Thirteenth AAAI*, volume 1, pages 181–187, 1996.

[155] Chris VOUDOURIS. « *Guided local search for combinatorial optimisation problems* ». PhD thesis, Department of Computer Science, University of Essex, Colchester, UK, 1997.

[156] Joachim Paul WALSER. « *Domain-independent local search for linear integer optimization* ». PhD thesis, Technische Fakultät der Universität des Saarlandes, 1998.

[157] Hang XU, Zhi-Long CHEN, Srinivas RAJAGOPAL, et Sundar ARUNAPURAM. « Solving a Practical Pickup and Delivery Problem ». *Transportation Science*, 37(3) : 347–364, 2003.

[158] Jeifeng XU et James P. KELLY. « A network flow-based tabu search heuristic for the vehicle routing problem ». *Transportation Science*, 30(4) : 379–393, 1996.

[159] M. YGIURA et T. IBARAKI. « The use of dynamic programming in genetic algorithms for permutation problems ». *European Journal of Operational Research*, 92 : 387–401, 1996.

[160] Tallys H. YUNES, Arnaldo V. MOURA, et Cid C.DE SOUZA. « Hybrid column generation approaches for solving real world crew management problems ». *Lecture Notes in Computer Science*, 1753 : 293–332, 2000.