

**Génération de Colonnes et de Coupes  
utilisant des sous-problèmes  
de plus court chemin**

THÈSE DE DOCTORAT  
Spécialité : Informatique

ÉCOLE DOCTORALE D'ANGERS

Présentée et soutenue publiquement

le : 23 mai 2003

à : Angers

par : Alain Chabrier

Devant le jury ci-dessous :

Alexandre CAMINADA (Examineur), Docteur, France Telecom R&D

Brigitte JAUMARD (Rapporteur), Professeur, Université de Montréal

Michel MINOUX (Président du Jury), Professeur, Université Paris VI

Jean-Francois PUGET (Examineur), Docteur, ILOG R&D

François VANDERBECK (Rapporteur), Professeur, Université de Bordeaux 1

Directeur de Thèse : Jin-Kao HAO, Professeur, Université d'Angers

Laboratoire :

ILOG : 9, rue de Verdun, BP 85, 94253 Gentilly Cedex

LERIA : Université d'Angers, 2 bd Lavoisier, 49045 Angers cedex 01

## Remerciements

Voilà, trois ans (et un peu plus) ont passé. Même si il est toujours tentant de vouloir reprendre telle ou telle partie, il arrive un jour où il faut savoir mettre un point final et passer à une nouvelle étape. Ce moment est venu et je suis, malgré tout, satisfait du chemin parcouru et de l'objectif atteint.

Je tiens à remercier ici, et en m'excusant par avance pour les possibles oublis, toutes les personnes sans lesquelles ce travail n'aurait peut être jamais abouti ou même jamais commencé. Tout d'abord, mes parents, Jacqueline et Jean-Jacques, qui m'ont très tôt fait le cadeau (empoisonné?) de cette passion pour ce que nous pourrions résumer sous l'étiquette de "monde de l'optimisation" ; plus tard, Jean-François Puget qui m'a fait confiance et m'a permis de faire se rejoindre cette passion et mon activité professionnelle. Je voudrais également remercier toutes les personnes d'ILOG qui ont participé, par leurs discussions et leurs actions volontaires et involontaires, à la maturation de ce travail et de son auteur. Les discussions interminables avec Javier Lafuente ont été (et resteront) parmi les meilleurs moments de ces années. Je remercie également Kathleen Callaway pour son aide lors de la traduction à l'anglais de certains articles ainsi que pour ses conversations qui m'ont souvent aidé à mieux comprendre la mentalité anglo-saxonne.

Aussi, je tiens à remercier tout particulièrement Jin-Kao Hao, qui a accepté de diriger cette thèse au contenu et méthodes pas toujours très "académiques". Enfin, je remercie les autres membres du jury, Alexandre Caminada, Michel Minoux et en particulier Brigitte Jaumard et François Vanderbeck pour l'intérêt qu'ils ont porté à mes travaux en acceptant d'en être les rapporteurs.

D'autre part, tout ceci n'aurait pas été possible si Mónica n'avait pas accepté que ce travail vienne (trop) souvent s'ajouter à l'autre et se retrancher aux moments de repos. Hugo, après avoir initialement refusé de respecter le planning, m'a ultérieurement laissé quelques moments de calme pour pouvoir ficeler les derniers détails.

# Table des matières

<b>Introduction</b>	<b>8</b>
<b>I État de l'art : Génération de Colonnes et de Coupes</b>	<b>13</b>
<b>1 Les méthodes de génération de colonnes</b>	<b>15</b>
1.1 Problèmes de satisfaction et d'optimisation . . . . .	16
1.1.1 Programmation Linéaire (PL) . . . . .	17
1.1.2 Programmation linéaire en nombres entiers . . . . .	19
1.1.3 Autres méthodes d'optimisation . . . . .	20
1.2 Méthodes de génération de colonnes . . . . .	21
1.2.1 Historique . . . . .	21
1.2.2 Idées de base de la génération de colonnes . . . . .	22
1.2.3 L'ajout retardé de colonnes au simplexe . . . . .	23
1.2.4 La décomposition de Dantzig-Wolfe . . . . .	23
1.2.5 Procédures de génération de colonnes . . . . .	26
1.2.6 Exemple illustratif . . . . .	28
1.3 Exemples académiques . . . . .	30
1.3.1 Découpe uni-dimensionnelle . . . . .	30
1.3.2 Tournées de véhicules . . . . .	34
1.3.3 Problèmes de plus court chemin . . . . .	40
1.4 Applications . . . . .	40
1.4.1 Tournées de véhicules et variantes . . . . .	41
1.4.2 Conception de réseau . . . . .	43
1.4.3 Planification de ressources . . . . .	44
1.4.4 Caractéristiques communes de ces problèmes . . . . .	46
<b>2 État de l'art</b>	<b>49</b>
2.1 Modèles . . . . .	49
2.2 Génération de colonnes simple . . . . .	50
2.2.1 Solution du problème relâché . . . . .	50

2.2.2	Solutions entières . . . . .	50
2.3	Branch-and-Price . . . . .	52
2.3.1	Dualité avec la méthode de <i>Branch-and-Cut</i> . . . . .	53
2.3.2	Règles de branchement . . . . .	54
2.3.3	Arbre de recherche et heuristiques . . . . .	57
2.4	Sous-problème de plus court chemin . . . . .	58
2.4.1	Le problème du <i>SPRCTW</i> . . . . .	58
2.4.2	Les algorithmes à base d'étiquettes . . . . .	59
2.4.3	Autres types de contraintes . . . . .	61
2.4.4	Algorithmes de programmation par contraintes . . . . .	61
2.5	Méthodes d'accélération . . . . .	63
2.5.1	Méthodes d'accélération du problème maître . . . . .	64
2.5.2	Méthodes d'accélération du problème de plus court chemin seul. . . . .	71
2.5.3	Méthodes d'accélération mettant en jeu les deux problèmes	73
2.6	Coupes : <i>Branch-and-Price-and-Cut</i> . . . . .	74
2.6.1	Coupes pour le problème relâché . . . . .	75
2.6.2	Coupes sur la validité . . . . .	75
2.6.3	Gestion des coupes en génération de colonnes . . . . .	76
<b>II</b>	<b>Modélisation et Recherche Générique</b>	<b>77</b>
<b>3</b>	<b>Modélisation générique</b>	<b>81</b>
3.1	Modèles . . . . .	81
3.1.1	Sous-problème . . . . .	81
3.1.2	Problème Maître . . . . .	83
3.1.3	Problème de Génération de Colonnes . . . . .	84
3.1.4	Coupes . . . . .	87
3.1.5	Branchement . . . . .	88
3.2	Exemples de modèles . . . . .	90
3.2.1	Le problème de découpe à une dimension . . . . .	90
3.2.2	Crew Scheduling Problem . . . . .	91
3.2.3	Crew Scheduling Problem avec équivalences . . . . .	92
3.2.4	Problème d'Affectation Généralisé . . . . .	93
<b>4</b>	<b>Recherche générique</b>	<b>95</b>
4.1	Utilisations antérieures des <i>goals</i> . . . . .	96
4.1.1	Les Goals en programmation par contraintes . . . . .	96
4.1.2	Les Goals dans les Problèmes Linéaires en Nombres Entiers	98
4.2	Les <i>goals</i> en génération de colonnes . . . . .	99

4.2.1	Goals pré-définis pour la génération de colonnes . . . . .	100
4.2.2	Branchement . . . . .	101
4.2.3	Stratégies de Recherche . . . . .	102
4.2.4	Informations accessibles aux goals . . . . .	103
4.2.5	Quand et comment éliminer un nœud? . . . . .	103
4.2.6	Quand considérer une solution entière comme valide? . . .	104
4.3	Exemples pour la génération de colonnes . . . . .	104
4.3.1	Générateur de colonnes . . . . .	104
4.3.2	Trouver l'optimum relâché . . . . .	105
4.3.3	Génération de colonnes simple . . . . .	105
4.3.4	<i>Branch-and-Price</i> . . . . .	105
4.3.5	<i>Branch-and-Price</i> avec recherche PLNE sur certains nœuds	106
4.3.6	Utilisation de différents générateurs . . . . .	106
4.3.7	Génération de colonnes partielle . . . . .	108
4.3.8	<i>Branch-and-Price-and-Cut</i> . . . . .	108

### III Applications 111

<b>5</b>	<b>Tournées de véhicules</b>	<b>115</b>
5.1	Le problème du plus court chemin avec fenêtres de temps pur en génération de colonnes . . . . .	116
5.2	Problèmes de VRP et VRPTW . . . . .	117
5.3	Autres extensions au VRP . . . . .	119
5.3.1	Flottes hétérogènes . . . . .	119
5.3.2	Dépôts multiples . . . . .	119
5.4	Problèmes avec cycles et sans cycles . . . . .	119
5.4.1	Cas du VRP . . . . .	120
5.4.2	Validité de l'utilisation de chemins non-élémentaires . . . .	120
5.5	Comparaison des bornes inférieures . . . . .	121
5.5.1	Exemple simple . . . . .	121
5.5.2	Utilisation de coupes . . . . .	121
5.5.3	Utilisation de chemins élémentaires . . . . .	122
5.6	Plus-court chemins élémentaires . . . . .	123
5.6.1	Une première règle de dominance modifiée . . . . .	124
5.6.2	Amélioration complète . . . . .	124
5.7	Améliorations heuristiques . . . . .	125
5.7.1	Réduction heuristique simple . . . . .	126
5.7.2	Niveaux de dominance . . . . .	126
5.7.3	Réductions heuristiques du graphe . . . . .	126
5.8	Autres améliorations dans l'implantation . . . . .	127

5.8.1	Mémoire d'étiquettes . . . . .	127
5.9	Autres améliorations . . . . .	127
5.9.1	Réductions du graphe utilisant les fenêtres de temps . . . . .	127
5.9.2	Nombre minimum de chemins . . . . .	128
5.9.3	Réductions basées sur le coût réduit . . . . .	128
5.10	Benchmark de Solomon . . . . .	128
5.11	Résultats . . . . .	129
<b>6</b>	<b>Planification de ressources</b>	<b>135</b>
6.1	Problématique des compagnies aériennes . . . . .	136
6.1.1	Les problèmes d'optimisation dans les compagnies aériennes	136
6.1.2	Génération de pairings . . . . .	138
6.1.3	Modèles de génération de colonnes . . . . .	141
6.1.4	Autres problèmes de planification de ressources . . . . .	144
6.2	Utilisation de schémas de coûts différents . . . . .	144
6.2.1	Format de coût utilisé en <i>Crew Scheduling</i> . . . . .	145
6.2.2	Coût-réduit . . . . .	145
6.2.3	Modification de l'algorithme . . . . .	145
6.3	Traitement des bases . . . . .	146
6.4	Contraintes additionnelles non compatibles . . . . .	147
6.4.1	Contraintes non compatibles . . . . .	147
6.4.2	Sous-problème avec PPC . . . . .	148
6.4.3	Heuristique d'expert . . . . .	155
6.4.4	Stratégies de recherche . . . . .	155
6.4.5	Résolution du Problème Maître . . . . .	156
6.5	Résultats . . . . .	158
6.5.1	Crew . . . . .	158
6.5.2	Viva . . . . .	158
6.6	Conclusions . . . . .	160
<b>7</b>	<b>Conception de réseau</b>	<b>163</b>
7.1	Le benchmark de conception de réseau . . . . .	165
7.2	Modèles de Génération de Colonnes . . . . .	167
7.2.1	Modèle PLNE . . . . .	167
7.2.2	Modèle décomposé . . . . .	168
7.2.3	Contraintes additionnelles . . . . .	170
7.2.4	<i>Génération de Colonnes Simple</i> . . . . .	171
7.2.5	<i>Branch-and-Price</i> . . . . .	172
7.3	Amélioration des bornes inférieures . . . . .	175
7.3.1	Description des coupes utilisées . . . . .	175
7.3.2	Coupes de type <i>Set</i> . . . . .	176

7.3.3	Coupes de type <i>BigDemand</i> . . . . .	178
7.3.4	Coupes de type <i>Connex</i> . . . . .	178
7.3.5	Coupes de type <i>Link</i> . . . . .	179
7.3.6	Détails de la recherche <i>Branch-and-Price-and-Cut</i> . . . . .	179
7.3.7	Résultats expérimentaux . . . . .	181
7.4	Amélioration des bornes supérieures . . . . .	183
7.4.1	Utilisation de la PLNE à certains nœuds . . . . .	183
7.4.2	Utilisation de stratégies de recherche . . . . .	184
7.4.3	Utilisation d’heuristique vers l’intégralité . . . . .	185
7.4.4	Elimination de nœuds . . . . .	185
7.5	Résultats expérimentaux . . . . .	186
7.5.1	Comparaison des bornes supérieures . . . . .	186
7.5.2	Influence de la limite de temps et des contraintes additionnelles . . . . .	189
7.6	Conclusions . . . . .	189
<b>A</b>	<b>Nos implantations : <i>Maestro</i> et <i>ShortestPath</i></b> . . . . .	<b>195</b>
A.1	<i>Maestro</i> . . . . .	195
A.1.1	IloMaestro . . . . .	196
A.1.2	IloBPColumn . . . . .	196
A.1.3	Goals . . . . .	197
A.1.4	IloBPBranchingRule . . . . .	197
A.1.5	IloBPCut . . . . .	199
A.1.6	IloBPNodeEvaluator . . . . .	199
A.1.7	Parallélisme . . . . .	199
A.2	<i>ShortestPath</i> . . . . .	200
A.3	Un exemple complet . . . . .	202
A.3.1	Fonction principale . . . . .	202
A.3.2	La classe de colonnes . . . . .	203
A.3.3	Création du problème maître . . . . .	204
A.3.4	Ajout des variables d’écart . . . . .	205
A.3.5	Goal de recherche . . . . .	205
A.4	Comparaisons avec d’autres systèmes . . . . .	208

## Introduction

Les problèmes d'optimisation combinatoire sont de plus en plus présents dans notre société. Durant les cinquante dernières années, de nombreuses méthodes ont été développées pour résoudre ces problèmes. Nous nous intéressons ici en particulier à certaines grandes familles de méthodes : la programmation linéaire en nombres entiers, la programmation par contraintes, la programmation dynamique et la recherche locale, celle-ci incluant les méta-heuristiques.

Au sein de la programmation linéaire, une technique, bien qu'ancienne, a récemment été le sujet de nombreuses publications : la génération de colonnes. Son champ d'application très varié et nous donnerons dans cette thèse des exemples pour les industries de la logistique, du transport aérien et des télécommunications.

Une littérature croissante au cours de ces dernières années a introduit diverses améliorations de la méthode originelle en les appliquant à certains problèmes particuliers. Ces études font intervenir des domaines de recherche très variés, raison probable, à notre avis, de la faible généralisation de ces résultats. En effet, le domaine de la programmation linéaire a apporté de nombreux résultats sur le traitement du problème maître en y ré-utilisant des techniques de la programmation linéaire standard. De même, la programmation dynamique a contribué à l'amélioration des algorithmes utilisés pour traiter les sous-problèmes, en particulier pour les cas où celui-ci est un problème de plus court chemin. Finalement, la programmation par contraintes et la recherche locale ont récemment eu des apports pour les applications de la génération de colonnes où les sous-problèmes ne permettaient plus l'utilisation de la programmation dynamique ou linéaire.

Cette thèse se propose de montrer que l'utilisation en génération de colonnes de nombreuses techniques provenant d'autres familles de méthodes de résolution peuvent facilement se généraliser à des problèmes variés. Pour cela, nous introduirons un formalisme applicable à l'ensemble des problèmes pouvant être résolus avec des méthodes de génération de colonnes avec sous-problème de plus court chemin. Ensuite, une méthode d'amélioration décrite selon ce formalisme pour un problème particulier pourra plus facilement être appliquée à d'autres problèmes. Plus concrètement, les contributions de cette thèse sont les suivantes.

Nous avons tout d'abord établi un inventaire des différentes méthodes d'accélération proposées pour certaines catégories particulières de problèmes. La présentation de chaque méthode est faite de manière indépendante de son application d'origine.

Ensuite, nous proposons un formalisme original permettant de décrire de manière générique les modèles de génération de colonnes. Nous proposons par ailleurs d'utiliser le formalisme de goal pour décrire de manière générique les procédures de résolution. La formulation des modèles et procédures de recherche des différents problèmes avec ces formalismes génériques permettra de leur appli-

quer aisément les différentes améliorations. Ces formalismes s'appliquent également aux nombreux cas où le sous-problème n'est pas un problème de plus court chemin.

Une implantation correspondant à ces deux formalismes ainsi qu'aux autres résultats présentés dans cette thèse a par ailleurs été développée. Deux bibliothèques C++ permettent le développement rapide de méthodes de génération de colonnes et de coupes avec sous-problème de plus court chemin.

Nous illustrons alors la possibilité de généraliser les méthodes d'accélération en proposant diverses améliorations pouvant être appliquées indistinctement à différents problèmes. Concrètement, les améliorations concernent les aspects suivants :

- l'utilisation d'algorithmes de plus court chemins modifiés pour les sous problèmes (chemins élémentaires, différents schémas de coûts, utilisation de la Programmation par Contraintes),
- une combinaison d'*heuristiques d'expert* et de programmation par contraintes pour le sous-problème,
- des stratégies de recherche pour le sous-problème,
- une contrainte globale de plus court chemin en programmation par contraintes pour le sous-problème,
- l'introduction de nouvelles coupes dans le problème maître,
- des heuristiques et stratégies de recherche dans le problème maître.

Ces améliorations sont enfin validées par la résolution de trois applications réelles de natures très différentes : la tournée de véhicules, la planification de ressources et la conception de réseau. Pour chacune des trois applications ont été implantés dans notre environnement des problèmes représentatifs pour lesquels nous avons obtenus des résultats prometteurs. En particulier, pour le VRP, nous avons fermé plusieurs instances réputées par une preuve d'optimalité, et pour la conception de réseau, nous présentons les meilleurs résultats publiés sur un benchmark public.

Cette thèse est organisée autour de 3 parties. La première introduit les multiples concepts mis en œuvre dans les méthodes de génération de colonnes, la deuxième propose un formalisme permettant la facile généralisation des méthodes utilisées en génération de colonnes, enfin la dernière présente trois types d'applications à des problèmes réels pour lesquelles nous présentons des contributions originales.

La première partie est composée de deux chapitres. Le premier chapitre est une introduction simple et didactique permettant de se familiariser avec les différents concepts intervenant ultérieurement. Après avoir introduit les notions de problèmes d'optimisation et présenté les grandes familles de méthodes existantes pour les résoudre, les idées de base de la génération de colonnes sont formalisées. Nous concluons ce chapitre en montrant comment et pourquoi différents modèles de

génération de colonnes peuvent inclure un sous-problème de plus court chemin et en illustrant la catégorie de problèmes correspondante par une brève présentation des trois types de problèmes sur lesquels nos contributions sont ultérieurement présentées. Le chapitre 2 présente un état de l'art sur les différentes améliorations apportées à la génération de colonnes durant ces dernières années, et en particulier aux cas où le sous-problème est un problème de plus court chemin.

La deuxième partie comprend deux chapitres introduisant les formalismes de description générique des modèles et procédures de recherche. Le premier (chapitre 3) introduit un formalisme permettant une modélisation générique des problèmes traitables par génération de colonnes. Le deuxième (chapitre 4) propose d'utiliser le concept de *goals* pour décrire leurs procédures de recherche.

Finalement, la dernière partie propose différentes contributions permettant des améliorations concrètes pour différents problèmes résolus par génération de colonnes avec sous-problèmes de plus court chemin. Celles-ci sont l'utilisation de sous-problème de plus court chemin élémentaire pour les problèmes de tournées de véhicules (chapitre 5), d'heuristiques pour la résolution des sous-problème de planification de ressources (chapitre 6), l'utilisation de coupes, d'heuristiques et de stratégies de recherche dans le problème maître de la conception de réseau (chapitre 7). A chacun de ces chapitres correspond un type d'application concret sur lequel nos contributions ont permis d'obtenir des résultats meilleurs que ceux jusqu'alors connus.

## Acronymes

BP	Branch&Price
BPC	Branch&Price&Cut
CSP	Constraint Satisfaction Problem
PL	Programmation Linéaire
PLNE	Programmation Linéaire en Nombres Entiers
PPC	Programmation par Contraintes
SPRCTW	Shortest Path with Resource Constraint and Time Windows
VRP	Vehicle Routing Problem
VRPTW	Vehicle Routing Problem with Time Windows

## Notations

$A$	Ensemble des arcs dans un graphe
$d$	Nœud de destination d'un chemin
$K$	Ensemble de véhicules
$N$	Nombre de nœuds dans un graphe
$o$	Nœud d'origine d'un chemin
$X$	Ensemble des nœuds dans un graphe



## Première partie

# État de l'art : Génération de Colonnes et de Coupes



# Chapitre 1

## Les méthodes de génération de colonnes

Les méthodes de génération de colonnes permettent de résoudre certains *problèmes d'optimisation* apparaissant dans de nombreux domaines de l'industrie. En particulier, cette thèse présente des exemples d'applications à la logistique, au transport aérien et aux télécommunications.

Ce chapitre introduit une formalisation très simple de ces problèmes d'optimisation, présente quelques idées générales sur les diverses familles de méthodes de résolution existantes, et introduit de manière didactique les méthodes de génération de colonnes (et en particulier celles avec sous-problème de plus court chemin) ainsi que les classes de problèmes sur lesquels elles peuvent être appliquées. En effet, les méthodes de génération de colonnes s'appliquent en particulier à certains problèmes linéaires contenant un nombre important de variables.

Ce chapitre est organisé comme suit. Dans la première section 1.1, les problèmes d'optimisation, et plus particulièrement les problèmes linéaires et problèmes linéaires en nombres entiers sont présentés ainsi que les méthodes de résolution qui leur sont associées. Nous donnons également un bref aperçu d'autres types de problèmes et méthodes de résolution. La section 1.2 introduit ensuite les méthodes de génération de colonnes et les positionne vis-a-vis des méthodes générales de programmation linéaire en nombres entiers. Puis, la section 1.3 présente deux exemples académiques illustratifs : le problème de la découpe uni-dimensionnelle et le problème de tournées de véhicules. Ce dernier nous permet d'introduire la sous famille des problèmes pour lesquels le sous-problème de génération de colonnes est un problème de plus court chemin. La section 1.4 conclut ce chapitre avec une présentation générale des trois problèmes réels sur lesquels sont présentées nos contributions ainsi qu'une discussion sur les caractéristiques communes à ces problèmes.

## 1.1 Problèmes de satisfaction et d'optimisation

Les problèmes apparaissant dans l'industrie sont étudiés en utilisant les concepts de contraintes, variables contraintes, domaines de variables, et sont rangés en deux catégories : les *problèmes de satisfaction de contraintes* et les *problèmes d'optimisation*. Nous introduisons ici brièvement ces définitions. Une formalisation plus complète est proposée dans [Tsa93].

**Définition 1** Une variable numérique  $x_i$  de domaine  $d_i$  est une variable ne pouvant prendre qu'une valeur appartenant à l'ensemble de valeurs  $d_i$ .

Dans le domaine de la programmation par contraintes, ce concept a été étendu aux variables prenant un ensemble de valeurs.

**Définition 2** Une contrainte  $C = (X_C, R_C)$  est définie par un ensemble de variables  $X_C \subseteq X$  et une relation  $R_C$  entre les variables de  $X_C$  :  $R_C \subseteq \prod_{x_i \in X_C} d_i$ . Cette relation représente les combinaisons permises pour les valeurs des domaines des variables.

**Définition 3** Un réseau de contraintes est représenté par un triplet  $(X, D, C)$  tel que :

- $X = \{x_1, \dots, x_n\}$  est un ensemble de variables,
- $D = \{d_1, \dots, d_n\}$  est l'ensemble des domaines. A toute variable  $x_i$  de  $X$  est associé le domaine fini  $d_i$ ,
- $C$  un ensemble de contraintes.

**Définition 4** Une affectation  $A$  de  $X_A \subseteq X$  est une application qui associe à toute variable  $x \in X_A$  une valeur  $v$  de son domaine. Une affectation est dite complète si  $X_A = X$ .

On dit qu'une contrainte  $C$  est complètement affectée par  $A$  si toutes les variables de  $X_C$  sont affectées (i.e.  $X_C \subseteq X_A$ ). On dit qu'une contrainte  $C$  est satisfaite par  $A$  si elle est complètement affectée et si les valeurs affectées aux variables de  $X_C$  satisfont la relation  $R_C$ . On dit qu'une affectation est cohérente si toutes les contraintes affectées sont satisfaites.

**Définition 5** Un problème de satisfaction de contraintes (CSP) est le problème consistant à déterminer si un réseau de contraintes admet une affectation complète et cohérente.

**Définition 6** Un objectif  $O$  sur les variables de  $X$  est une paire  $O \equiv (e(X), s)$  où  $e(X)$  est une expression arithmétique sur les variables de  $X$  et  $s$  un sens d'optimisation avec  $s \in (\text{minimiser}, \text{maximiser})$ .

**Définition 7** Un problème d'optimisation  $OP$  est un triplet  $(X, C, O)$ , où  $X$  est un ensemble de variables,  $C$  un ensemble de contraintes et  $O$  un objectif.

Remarquons qu'un problème de satisfaction peut être mis sous forme d'un problème d'optimisation en supprimant certaines contraintes et en les intégrant à l'objectif à optimiser. Le problème de satisfaction revient alors à minimiser le nombre de contraintes non vérifiées, la satisfaction étant obtenue quand l'objectif de 0 est atteint.

### 1.1.1 Programmation Linéaire (PL)

Parmi tous les problèmes d'optimisation pouvant être définis avec ce formalisme, une catégorie, la *Programmation Linéaire*, traite des problèmes ne comportant que des contraintes consistant à borner des expressions linéaires. Cette famille de problèmes et méthodes nous intéresse particulièrement car elle englobe les méthodes de génération de colonnes. Pour une description complète des méthodes de programmation linéaire, un ouvrage de référence comme [Chv] peut être consulté.

#### Définitions

**Définition 8** Une contrainte linéaire  $CL$  est un quadruplet  $(Y, e_{CL}(Y), lb, ub)$  où  $Y$  est un ensemble de variables (éventuellement vide),  $e_{CL}(Y)$  une expression linéaire de ces variables et  $lb$  et  $ub$  des valeurs réelles représentant des bornes inférieure et supérieure pour cette expression. Notons que  $lb$  peut être  $-\infty$  et  $ub$  peut être  $+\infty$ . Quand la contrainte utilise un ensemble de variables initialement vide, elle peut être abrégée  $CL \equiv (lb, ub)$ .

**Définition 9** Un objectif linéaire  $OL$  est un triplet  $(Y, e_{OL}(Y), s)$  où  $Y$  est un ensemble de variables (éventuellement vide),  $e_{OL}(Y)$  une expression linéaire de ces variables et  $s$  un sens d'optimisation. Quand l'objectif linéaire utilise un ensemble de variables vide, nous le noterons  $OL \equiv (s)$ .

Un problème linéaire ne contient alors que des contraintes linéaires et au plus un objectif linéaire. De nombreuses études ont été réalisées sur la résolution de ce problème. La méthode dite du *simplex* permet en pratique de résoudre efficacement ce problème sur des variables continues (i.e. dont les domaines sont des intervalles de  $\mathbb{R}$ ).

La correspondance directe entre les variables du problème linéaire et les colonnes de la matrice définissant les contraintes a pour conséquence un usage alternatif des mots variables et colonnes en fonction du contexte. Ces deux termes sont équivalents dans le cadre de la programmation linéaire. La même équivalence

existe entre les contraintes du problème et les lignes de la matrice, cette dernière n'étant en fait qu'une représentation particulière d'un ensemble de contraintes.

### Programme Linéaire

Un *programme linéaire* est un problème d'optimisation sur un ensemble de variables  $x = (x_i)$ , ayant une fonction objectif de la forme :

$$\max \sum_i c_i x_i$$

ou

$$\min \sum_i c_i x_i$$

ainsi que des contraintes d'une des formes suivantes :

$$A_1 x \leq b_1$$

$$A_2 x = b_2$$

$$A_3 x \geq b_3$$

Une *forme standard* existe où l'objectif est toujours de maximisation et où toutes les contraintes sont des contraintes d'égalité. Un programme linéaire peut toujours être mis sous une *forme canonique* standard où chaque contrainte possède une variable qui lui est propre de coefficient 1.

### Méthode du Simplex

La méthode du simplex par tableaux permet de résoudre un problème donné dans sa forme canonique. Une *base*, i.e. une sous-matrice carrée régulière extraite de la matrice complète, est utilisée. L'algorithme contient 5 étapes :

1. Initialisation : trouver une base réalisable,
2. Test d'optimalité : teste si la solution courante est optimale. Si elle est optimale, l'algorithme se termine.
3. Variable hors base entrante : dans le cas contraire, choisir une variable hors base appelée à devenir une variable de base,
4. Variable de base sortante : choisir la variable de base cédant sa place,
5. Pivotage : échange de ces deux variables. Mise à jour du tableau. Retour à l'étape 2.

La méthode du simplex par tableaux utilise, comme son nom l'indique, un tableau dont le contenu est actualisé à chaque itération de l'algorithme.

### Méthode révisée du Simplex

L'algorithme révisé du simplex n'utilise pas de tableaux. A chaque itération, seules les informations spécifiant la base courante sont stockées. Les autres grandeurs décidant de la poursuite de l'algorithme ne sont calculées qu'au besoin. Une telle approche permet des gains de temps et d'effort, parfois importants, par rapport aux méthodes calculant entièrement les tableaux. La procédure se résume alors à :

1. Calcul des solutions de base primale et duale.
2. Recherche une variable entrante : Parcourir les variables hors base, et pour chacune d'elles, calculer son coût réduit. Si aucun coût réduit négatif n'est obtenu, les solutions de base actuelles sont optimales ; sinon, la première variable possédant un coût réduit négatif entre dans la base.
3. Recherche d'une variable sortante.
4. Mise à jour.

Le fait de ne garder aucune information sur les variables hors base prend toute son importance quand le nombre de variables est très grand. Des méthodes de génération de colonnes sont alors possibles. En effet, l'itération 2 de la méthode révisée revient alors à parcourir les variables hors base à la recherche d'une variable de coût réduit favorable.

### 1.1.2 Programmation linéaire en nombres entiers

Les problèmes réels concernent souvent des variables qui ne peuvent pas être considérées continues, (i.e. dont les domaines de valeurs possibles sont des sous ensembles de  $\mathbb{N}$ ). La *Programmation Linéaire en Nombres Entiers* (PLNE) fournit alors des outils permettant de respecter ces restrictions supplémentaires.

La méthode du simplex permet de résoudre des programmes linéaires donnés sous forme canonique sur des variables continues non bornées. Des variantes de la méthode ont été proposées qui permettent de prendre en compte des bornes finies pour les variables autrement que par l'ajout explicite de contraintes. Cependant, certaines variables sont fréquemment restreintes à ne prendre que des valeurs entières. La méthode du simplex ne permet pas de traiter directement ces contraintes d'intégralité.

Afin d'obtenir la solution optimale d'un programme linéaire en nombres entiers, une recherche arborescente doit être mise en œuvre. A chaque nœud, une borne est obtenue sur le problème local courant via l'utilisation de la méthode du simplex. Si cette borne est plus mauvaise que la meilleure solution entière trouvée jusqu'alors, la branche courante est abandonnée. Sinon, si la solution est entière, nous avons alors une nouvelle meilleure solution primale. Enfin, si la solution

n'est pas entière, il faut effectuer un branchement par séparation. La méthode la plus simple et la plus fréquemment utilisée consiste à choisir une des variables entières  $x_i$  de valeur fractionnelle  $x_i^*$  dans la solution courante et à brancher en ajoutant à chacune des deux sous branches respectivement :

$$x_i \geq \lceil x_i^* \rceil \text{ et } x_i \leq \lfloor x_i^* \rfloor$$

Dans chacune des branches, la solution courante est éliminée par le problème plus contraint.

### Coupes

Afin d'éliminer les solutions relâchées non entières, il n'est pas toujours nécessaire de brancher. En effet, de nouvelles contraintes qui éliminent la solution relâchée courante mais qui n'éliminent pas la solution entière, peuvent être ajoutées. Ces contraintes sont appelées *coupes* en raison de leur représentation graphique. Elles coupent en effet un point extrême du polyèdre courant, sans pour autant éliminer les points entiers intérieurs à ce polytope. L'utilisation de ces coupes de manière itérative permet théoriquement d'atteindre la solution entière optimale. Dans la pratique, cela peut se révéler extrêmement inefficace.

### Application aux problèmes avec beaucoup de contraintes

Parfois, le nombre de contraintes linéaires est trop important pour que toutes soient prises en compte directement. Une méthode consiste alors à n'utiliser initialement que certaines d'entre elles. Une solution entière peut alors apparaître qui ne soit pas valide pour l'ensemble du problème, i.e. une solution qui *viole* l'une des contraintes non explicitement ajoutées au problème. Ces contraintes violées, ici aussi appelés *coupes*, sont alors ajoutées explicitement.

### 1.1.3 Autres méthodes d'optimisation

De nombreuses autres méthodes permettent d'isoler la solution optimale d'un problème d'optimisation. En plus de la programmation linéaire, nous utilisons des concepts provenant de trois autres grandes familles de méthodes de résolution, chacune s'appliquant à des problèmes ayant des caractéristiques particulières. Ces familles sont celles de la *Programmation Dynamique*, la *Programmation par Contraintes*, et la *Recherche Locale*. D'autres familles de méthodes existent encore, par exemple dans le domaine des problèmes non linéaires.

Le famille des méthodes de *Programmation Dynamique* tient sa cohérence non pas de la forme des problèmes, comme pour la Programmation Linéaire, mais d'un concept commun à toutes les méthodes de résolution s'y attachant. Ce principe de

base, connu sous le nom de principe d’optimalité de Bellman, pourrait s’énoncer simplement ainsi : la restriction d’une solution optimale à une sous partie du problème est également optimale vis à vis du sous problème correspondant au problème restreint à cette sous partie. Ce principe donne lieu à des méthodes utilisant la résolution de plusieurs problèmes réduits pour résoudre le problème complet. Ces méthodes sont largement décrites dans [Bel57].

La *Programmation par Contraintes* utilise des domaines associés aux variables. Des algorithmes de filtrage sont utilisés pour réduire les domaines de ces variables en établissant une cohérence locale. Quand aucune nouvelle réduction n’est possible, une variable est instantiée à une des valeurs possibles de son domaine. Cette modification est propagée en utilisant de nouveaux les algorithmes de filtrage. Une recherche arborescente permet alors d’obtenir des affectations complètes. Une introduction plus complète à la programmation par contraintes est disponible dans [MS98].

Les méthodes de *Recherche Locale* modifient localement une affectation complète. Diverses méthodes heuristiques et méta-heuristiques définissent comment cette affectation est modifiée. Ces méthodes ne permettent pas de prouver l’optimalité d’une solution, mais sont réputées comme permettant d’obtenir rapidement des solutions cohérentes. Un inventaire des méthodes est donné dans [Ree95].

## 1.2 Méthodes de génération de colonnes

La famille des méthodes de génération de colonnes est à l’origine une sous-famille de la programmation linéaire. En effet, elle s’appliquait initialement uniquement à des problèmes linéaires décomposés faisant apparaître de nombreuses variables. Une partie du problème décomposé peut cependant ne pas être linéaire, ce qui lui a donné une certaine indépendance vis à vis de la programmation linéaire et a attiré l’attention de personnes extérieures à ce domaine.

### 1.2.1 Historique

En 1958, Ford and Fulkerson [FF58] ont été les premiers à suggérer une méthode de type génération de colonnes, pour un problème de flux maximal dans un réseau pour plusieurs commodités. Par la suite, en 1960, Dantzig et Wolfe [DW60] formalisèrent une technique de décomposition applicable à des problèmes linéaires possédant une structure particulière. Ces deux travaux fondateurs posèrent les bases des deux idées centrales aux méthodes de génération de colonnes.

Puis, en 1961, Gilmore et Gomory [GG61] appliquèrent une combinaison de ces deux idées à un problème de découpe. C’est cette dernière étude qui reste

la plus connue, car elle propose une méthode qui allait être utilisée, sans grande modification durant les 35 années suivantes.

Finalement, les méthodes de génération de colonnes resteront presque oubliées jusque dans le milieu des années 1980. De nombreuses améliorations ont alors été proposées et de nos jours, ces méthodes sont utilisées dans de nombreux projets industriels.

Actuellement, en plus des problèmes présentés dans cette thèse, de nombreuses autres problèmes ont été résolus en utilisant des méthode de génération de colonnes. Par exemple :

- *Multicommodity Flow Problem* [BHV00], [VAK98],
- *Facility Location Problem* [KD02],
- *Coloration de graphes* [MT96].
- *Combinatorial Auctions Problem* [DV00],
- *Problèmes de découpe* [VBJN94], [DP98], [DS99], [Van98], [Van99].

### 1.2.2 Idées de base de la génération de colonnes

La terminologie *génération de colonnes* comprend communément un ensemble de techniques permettant de résoudre un problème en utilisant deux problèmes simultanément :

- un problème maître, linéaire,
- un sous-problème ayant pour rôle de générer de nouvelles colonnes valides utilisées comme variables du problème maître.

Deux idées sont au centre des méthodes de génération de colonnes : l'ajout retardé de colonnes dans la méthode révisée du simplexe et la décomposition de Dantzig-Wolfe. Ces deux idées sont en fait indépendantes et complémentaires.

Notons par ailleurs que l'application des méthodes de génération de colonnes ne nécessite pas la compréhension complète de ces deux idées et beaucoup d'utilisations actuelles des techniques de génération de colonnes se font sans avoir une connaissance exacte de ces deux fondements théoriques.

La première idée est en fait une caractéristique simple de la méthode révisée du simplexe utilisée en programmation linéaire : cette méthode ne nécessite pas un accès direct à l'ensemble des colonnes présentes dans le problème linéaire. A chaque itération, seul un accès ou bien à au moins une des variables hors base ayant un coût réduit favorable ou bien la preuve de non existence d'une telle variable sont suffisants. L'ensemble de l'algorithme du simplexe peut s'effectuer sans qu'à aucun moment ne soit utilisée en aucune façon une grande partie des variables du problème. Cette idée explique pourquoi les techniques de génération de colonnes s'appliquent aux problèmes linéaires avec un très grand nombre de variables : la majorité de ces variables ne vont pas être utilisées explicitement.

La deuxième idée est la *décomposition de Dantzig-Wolfe* que nous présentons de manière détaillée dans une prochaine section. Celle-ci permet de décomposer un problème ayant une partie linéaire et une partie non linéaire en deux problèmes interagissant <sup>1</sup>. Cette idée explique pourquoi les techniques de génération de colonnes s'appliquent particulièrement à des problèmes ayant une partie linéaire et une partie comprenant des contraintes plus complexes.

Finalement, notons que la liaison entre ces deux idées vient du fait que la méthode de décomposition de Dantzig-Wolfe a pour résultat de faire intervenir un très grand nombre de variables dans le problème maître du modèle décomposé. Celui-ci peut alors être résolu plus facilement en utilisant l'ajout retardé de certaines colonnes.

Souvent, la méthode de décomposition est oubliée, et le problème est directement présenté sous sa forme décomposée. Ceci est en particulier le cas quand ce modèle décomposé correspond directement à une réalité.

### 1.2.3 L'ajout retardé de colonnes au simplexe

A l'origine, la méthode du simplexe utilise un tableau qui est actualisé à chaque changement de base réalisé. La méthode révisée du simplexe permet d'éviter de conserver (et donc d'actualiser) ce tableau. L'étape de la méthode servant à trouver la colonne hors base devant entrer en base consiste, dans le cas où toutes les colonnes sont disponibles, à itérer sur ces colonnes et rechercher la première d'entre elles ayant un coût réduit favorable. La seule interaction existant alors entre la base courante et l'ensemble de toutes les autres colonnes candidates ne s'effectue que via un *oracle* qui, quand il est appelé par la procédure, est capable de retourner une colonne de coût réduit favorable ou bien d'assurer qu'il n'en existe aucune.

Cette caractéristique est centrale dans le fonctionnement des méthodes de génération de colonnes. En effet, celles-ci, au lieu de parcourir l'ensemble des colonnes hors base, mais existantes, utiliseront un nombre limité de colonnes ainsi qu'un oracle plus évolué capable de *générer une colonne* en fonction de certaines caractéristiques (contraintes) définissant les colonnes valides. Ces contraintes donnant forme aux colonnes peuvent apparaître naturellement ou bien résulter de la décomposition de Dantzig-Wolfe.

### 1.2.4 La décomposition de Dantzig-Wolfe

Nous présentons ici la décomposition de Dantzig-Wolfe de manière similaire à [Erd99]. Une présentation très complète est donnée dans [Van00].

---

<sup>1</sup>Notons que la méthode a été initialement introduite en utilisant un modèle avec deux parties toutes deux linéaires.

Considérons le problème linéaire :

$$\begin{aligned} \min \quad & c x \\ & Ax \leq b \\ & Bx \leq d \\ & x \geq 0 \end{aligned}$$

où  $c, x, b$ , et  $d$  sont des vecteurs,  $A$  et  $B$  des matrices. Nous allons remplacer  $Bx \leq d$  par des variables. Ceci peut donner un problème contenant énormément de variables pour être résolu directement. C'est pourquoi nous utiliserons ensuite des méthodes de génération de colonnes à la demande.

Remarquons que  $P = \{x \in \mathbb{R} \mid Bx \leq d, x \geq 0\}$  définit un polyèdre. Rappelons qu'un *polyèdre* est, par définition, l'intersection d'un nombre fini de demi-espaces et qu'un *polytope* est un polyèdre borné. Pour simplifier, nous supposons ici que  $P$  est un polytope. Un point de  $P$  peut alors être écrit comme une combinaison convexe des points extrêmes  $x^1, \dots, x^q$ , i.e. pour  $x \in P$ , il existe un ensemble  $\Omega$  de coefficients  $\lambda^1, \dots, \lambda^q$  tels que :

$$\begin{aligned} x &= \sum_{i=1}^q \lambda^i x^i \\ \sum_{i=1}^q \lambda^i &= 1 \\ \lambda^i &\geq 0, \quad \forall i \in \{1, \dots, q\} \end{aligned}$$

En substituant  $x$  dans le problème initial, nous obtenons alors un nouveau problème pour lequel les  $\lambda^i$  prennent le rôle de variables. En effet, chercher des valeurs pour le vecteur variable  $x$  revient à chercher des valeurs pour ses coefficients dans la combinaison linéaire des  $x^i$ , et donc des valeurs pour les "variables"  $\lambda^i$  :

$$\begin{aligned} \min \quad & \sum_{i=1}^q (cx^i) \lambda^i \\ & \sum_{i=1}^q (Ax^i) \lambda^i \leq b \\ & \sum_{i=1}^q \lambda^i = 1 \\ & \lambda^i \geq 0, \forall \lambda^i \in \Omega \end{aligned}$$

Les deux problèmes sont équivalents. Cette dernière formulation est le *problème maître de Dantzig-Wolfe*. Il existe une variable  $\lambda^i$  pour chaque point extrême du polytope  $P$  correspondant à la matrice de contraintes  $B$ . Cet ensemble  $\Omega$  de variables est en général trop grand pour que toutes les variables soient directement prises en compte. Nous appliquons alors la méthode révisée du simplexe au *problème maître restreint* qui est défini par :

$$\begin{aligned} \min \quad & \sum_{i \in \Omega^*} (cx^i) \lambda^i \\ & \sum_{i \in \Omega^*} (Ax^i) \lambda^i \leq b \\ & \sum_{i \in \Omega^*} \lambda^i = 1 \\ & \lambda^i \geq 0, \forall \lambda^i \in \Omega^* \end{aligned}$$

avec  $\Omega^*$  un sous-ensemble de  $\Omega$ . Nous supposons que le sous-ensemble est choisi tel que le problème restreint initial soit faisable. Soient  $(\pi, \pi_0)$  les valeurs duales associées avec les deux séries de contraintes précédentes. Le coût réduit  $rc_i$  d'une variable  $\lambda^i$  est alors  $rc_i = (c - \pi A)x^i - \pi_0$ . Si  $rc_i \geq 0$  pour tous les  $\lambda^i \in \Omega$ , la solution optimale du problème maître restreint est la solution optimale au problème maître complet. Si l'une des colonnes  $\lambda^i \in \Omega$  a un coût réduit favorable et n'est pas dans le problème maître restreint ( $\lambda^i \notin \Omega^*$ ), cette colonne doit lui être ajoutée. Le *sous-problème de Dantzig-Wolfe* est alors défini comme :

$$\begin{aligned} \min \quad & (c - \pi A)x - \pi_0 \\ & Bx \leq d \\ & x \geq 0 \end{aligned}$$

### Matrices Bloc angulaires

Les résultats précédents s'étendent aux cas de matrices bloc angulaires de la forme :

$$\begin{aligned} \min \quad & \sum_{k \in K} c_k x_k \\ & \sum_{k \in K} A_k x_k \leq b \\ & B_k x_k \leq d_k, \forall k \in K \end{aligned}$$

A chaque matrice  $B_k$  sera associé un sous-problème.

### Non-Linéarité

Dans le développement précédent, le coût  $c(x)$  avait la forme  $c(x) = cx$ . Notons cependant qu'il n'est pas nécessaire que la fonction de coût dans le problème d'origine soit linéaire. La non-linéarité est alors gérée dans le sous-problème. D'autre part, il est possible que le sous-problème ne soit pas linéaire. En effet, le problème à décomposer peut prendre la forme :

$$\begin{aligned} \min c(x) \\ Ax \leq b \\ x \in X \end{aligned} \tag{1.1}$$

On applique alors la décomposition de  $x$  suivant la base définissant  $\text{conv}(X)$ . Celle-ci peut s'avérer un peu plus complexe mais les résultats restent globalement similaires.

### 1.2.5 Procédures de génération de colonnes

Nous avons vu les deux idées centrales des méthodes de génération de colonnes que sont l'ajout retardé de colonnes pour la méthode révisée du simplexe et la décomposition de Dantzig-Wolfe. Nous nous référons aux techniques de génération de colonnes comme étant celles où ces deux idées interviennent simultanément. Comme nous l'avons indiqué précédemment, la décomposition n'est quasiment jamais présentée dans les articles et n'est souvent utilisée que de manière implicite.

Dans tous les cas, la génération de colonnes fait intervenir les deux problèmes introduits dans la section précédente, le problème maître restreint et le sous-problème tels qu'ils sont obtenus avec la décomposition. Que le modèle soit le résultat d'une décomposition ou qu'il apparaisse naturellement sous cette forme, la génération de colonnes fera toujours intervenir deux problèmes (au minimum) :

- un problème maître restreint, linéaire.
- un (ou plusieurs) sous-problème.

La figure 1.1 montre la structure centrale de tous les algorithmes utilisant les techniques de génération de colonnes. L'ensemble des procédures fonctionnent autour d'un échange d'information entre le problème maître restreint et le (ou les) sous-problème(s). Dans un sens, les valeurs duales correspondant à l'état actuel du problème maître sont passées au sous-problème qui en retour donne de nouvelles colonnes de coût réduit favorable.

### Résolution du problème relaxé

Nous avons jusque là volontairement évité d'indiquer avec précision la nature des variables (entières ou simplement continues). Dans le cas précédent où la

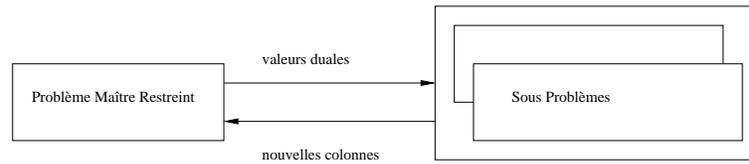


FIG. 1.1 – Structure centrale des algorithmes de génération de colonnes

matrice  $B$  définit un polytope, il est facile de voir qu'une solution où les  $\lambda^i$  ne sont pas entiers peut être valide pour le problème d'origine. Ceci ne sera pas toujours le cas (en particulier si la définition du polytope est changée en  $Bx = d$ ).

Dans le cas où les colonnes non présentes initialement et qui sont résultats de la résolution d'un sous-problème sont continues, la méthode révisée du simplexe permet, comme nous l'avons vu dans la section 1.2.3, de ne travailler que sur un problème restreint et cependant assurer l'optimalité de la solution finale sur le problème complet.

La structure présentée dans la figure 1.1 prend alors tout son sens. Le sous-problème est exécuté chaque fois que la méthode du simplexe aurait estimé le coût réduit des variables non en base. Si une colonne est trouvée avec un coût réduit favorable (i.e. négatif en cas de minimisation), alors elle est ajoutée au problème maître restreint, et la méthode du simplexe est poursuivie. Si aucune colonne de coût réduit favorable n'est trouvée, alors la solution de base du problème restreint est solution du problème complet.

Tout comme pour la méthode du simplexe normale, l'ordre dans lequel sont ajoutées les colonnes ne change pas la valeur de la solution finalement obtenue. Dans les cas de dégénérescence, des solutions différentes de même coût peuvent cependant être obtenues. D'autre part, le nombre d'itérations entre le problème maître restreint et le sous-problème peut être très différent suivant l'ordre dans lequel sont ajoutées les colonnes. La seule condition réellement intangible est la complétude de la résolution du sous-problème : si au moins une colonne de coût réduit favorable non présente dans le problème restreint existe, au moins une colonne de coût réduit favorable doit être trouvée. Sans cela, il est impossible d'assurer l'optimalité de la solution du problème restreint vis-à-vis du problème complet.

### Problème de l'intégralité

Dans le cas où les colonnes ne sont pas continues, la procédure est plus compliquée. La solution trouvée en relaxant l'intégralité et en utilisant la procédure que nous venons de décrire nous permet simplement de trouver une solution optimale au problème relaxé. Notons cependant que cette solution est une borne

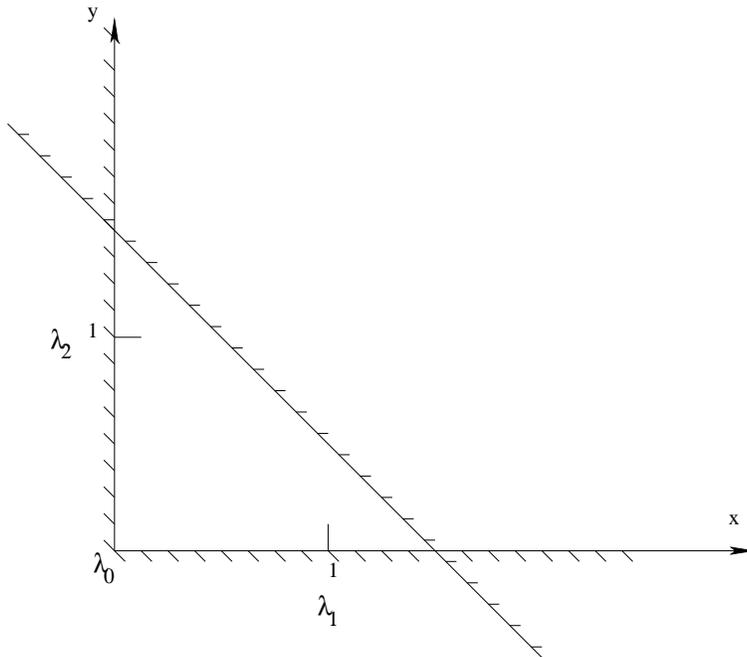


FIG. 1.2 – Exemple illustratif

inférieure pour la solution optimale entière au problème complet.

Quand il ne s'agit plus simplement de trouver une borne inférieure, mais une solution entière au problème complet, la procédure à suivre est plus complexe et moins évidente. Un large éventail de possibilités existe offrant des résultats allant de la simple recherche de solution sans garantie jusqu'à des méthodes arborescentes conservant la propriété de complétude. Nous reviendrons plus complètement sur ces différentes approches dans le prochain chapitre.

### 1.2.6 Exemple illustratif

Dans cette section, nous allons utiliser un exemple extrêmement simple afin d'illustrer plusieurs aspects de la décomposition précédemment introduite.

L'exemple que nous allons utiliser est le suivant :

$$\begin{aligned} \max x + y \\ 2x + 2y \leq 3 \\ x, y \in \mathbb{N} \end{aligned}$$

Nous pouvons voir sur la figure 1.2 une représentation de ce problème très simple. Le polyèdre de la relaxation (où les contraintes d'intégralité sont omises)

correspond aux trois points  $(0, 0), (0, \frac{3}{2}), (\frac{3}{2}, 0)$ . L'optimum relâché vaut donc  $OPT_{rel} = \frac{3}{2}$ .

Appliquons maintenant la décomposition de Dantzig-Wolfe à ce problème. Nous utiliserons les matrices correspondant aux ensembles de contraintes  $A = \emptyset$  et  $X = \{(x, y) \in \mathbb{N}^2, 2x + 2y \leq 3\}$ . Nous avons trois points extrêmes  $x^0, x^1$  et  $x^2$ , auxquels nous associons respectivement les variables de  $\Omega = \{\lambda^0, \lambda^1, \lambda^2\}$  :

$$x^0 = (0, 0)$$

$$x^1 = (\frac{3}{2}, 0)$$

$$x^2 = (0, \frac{3}{2})$$

Nous pouvons prendre pour le problème restreint  $\Omega^* = \{\lambda^0\}$ . Le problème restreint est alors :

$$\begin{aligned} \max & 0 \\ & \lambda^0 \leq 1 \end{aligned}$$

Une solution est alors  $\lambda^0 = 1$ , de valeur 0.

Le problème dual (si nous associons la variable duale  $u$  à l'unique contrainte du problème) est :

$$\begin{aligned} \min & u \\ & u \geq 0 \end{aligned}$$

La solution est  $u = 0$  (de valeur 0 également).

Si nous calculons le coût réduit pour les colonnes de  $\Omega \setminus \Omega^*$ , nous avons :

$$RC(\lambda^1) = 1 - 0 = 1$$

$\lambda^1$  est donc ajouté au problème restreint.

Nous avons donc :

$$\begin{aligned} \max & \lambda^1 \\ & \lambda^0 + \lambda^1 \leq 1 \end{aligned}$$

Le problème dual devient :

$$\begin{aligned} \min & u \\ & u \geq 0 \\ & u \geq 1 \end{aligned}$$

et sa solution est  $u = 1$ . Le coût réduit de  $\lambda^2$ , seule colonne susceptible d'être ajoutée est nul. Nous avons terminé. L'optimum relâché de la décomposition vaut 1. Notons qu'il est meilleur que celui du problème initial. Ce sera toujours le cas, car le polytope du problème décomposé est toujours inclus dans le polytope du problème initial.

Ici la solution est  $\lambda^1 = 1 \in \mathbb{N}$ , la solution relâchée est entière, il n'est donc pas nécessaire de recourir à des méthodes arborescentes pour trouver une solution entière.

Nous avons à chaque moment supposé que les trois colonnes  $\lambda^i$  étaient connues explicitement. Dans le cas général, les  $\lambda^i$  sont plus nombreuses, et ne sont obtenues qu'en résolvant le sous-problème qui est ici :

$$\begin{aligned} \max \quad & x + y - \pi_u \\ & 2x + 2y \leq 3 \\ & x, y \in \mathbb{N} \end{aligned}$$

avec  $\pi_u$  la valeur duale, i.e. la valeur de la variable duale  $u$  à l'optimum.

## 1.3 Exemples académiques

Nous présentons ici quelques exemples déjà très documentés. Nous les utilisons comme illustration en raison de leur simplicité et représentativité. Le premier exemple, de découpe uni-dimensionnelle, est souvent utilisé pour introduire les méthodes de génération de colonnes. Le second, de tournée de véhicules est quand à lui représentatif des problèmes avec sous-problème de plus court chemin.

### 1.3.1 Découpe uni-dimensionnelle

Après ce très simple exemple, nous allons illustrer l'application de la méthode de décomposition de Dantzig-Wolfe sur le problème qui historiquement fut utilisé dans le cadre de leurs premiers développements. Il s'agit d'un problème de découpe unidimensionnelle. Ce problème a l'avantage d'avoir une décomposition qui fait intervenir un sous-problème de forme très simple (problème de sac-à-dos unidimensionnel) et ainsi de ne pas associer systématiquement les méthodes de génération de colonnes aux sous-problèmes de plus court chemin. Cet exemple est par ailleurs suffisamment simple pour permettre une meilleure compréhension de la méthode de décomposition tout en étant déjà dans le cadre de son application à un problème réaliste. Enfin, cette application est intéressante car les méthodes de génération de colonnes sont véritablement parmi les méthodes les plus utilisées pour résoudre les problèmes réels correspondants.

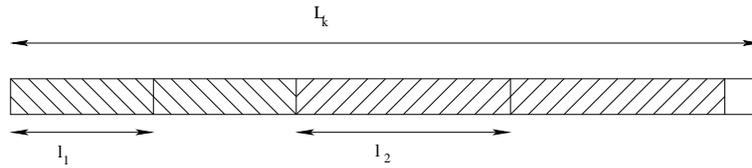


FIG. 1.3 – Exemple de patron de découpe.

Nous introduisons tout d’abord la problématique et proposons une formalisation du problème de découpe à une dimension et de minimisation du nombre de barres.

Nous commencerons par présenter un modèle de PLNE pour ce problème. Nous lui appliquons la décomposition de Dantzig-Wolfe afin d’obtenir un modèle de génération de colonnes. Nous discutons des différentes approximations du nombre de colonnes et présenterons une méthode de résolution simple correspondant à celle introduite initialement dans [GG61].

Nous discutons de la possibilité de n’obtenir aucune solution entière avec cette méthode. Nous discutons également la complétude de la méthode et la notion de borne inférieure globale.

### Le problème de découpe uni-dimensionnelle

Nous disposons de  $K$  barres de tailles  $L_k$ . Ces barres peuvent être découpées en petits bouts, appelés *éléments*, suivant une distribution définie par un *patron*. Les éléments pouvant être découpés sont de  $N$  tailles différentes notées  $l_1, \dots, l_N$ . Une demande minimale  $d_i$  est définie pour chaque type d’éléments. L’objectif est de trouver les patrons à utiliser pour couper les barres permettant de couvrir la demande en éléments tout en minimisant le nombre total de barres utilisées.

Sur la figure 1.3, nous voyons un exemple de patron, correspondant à la découpe de deux éléments de type 1 et deux éléments de type 2. Il reste un *déchet* (partie de barre non utilisée).

### Modèle de PLNE

Nous introduisons pour chaque barre  $k \in \{1, \dots, K\}$  des variables  $x_{ik}$  indiquant combien d’éléments de type  $i$  sont découpés dans la barre  $k$ . La contrainte sur la longueur maximale de barre disponible est donc :

$$\sum_{i=1}^N l_i x_{ik} \leq L_k, \forall k \in \{1, \dots, K\}$$

Une barre peut rester inutilisée, nous introduisons la variable  $u_k \in \{0, 1\}$  indiquant si la barre  $k$  est utilisée. La contrainte précédente devient alors :

$$\sum_{i=1}^N l_i x_{ik} \leq L_k u_k, \forall k \in \{1, \dots, K\}$$

Les contraintes sur les demandes sont :

$$\sum_{k=1}^K x_{ik} \geq d_i, \forall i \in \{1, \dots, N\}$$

Finalement, l'objectif est de minimiser le nombre de barres utilisées, soit :

$$\min \sum_{k=1}^K u_k \tag{1.2}$$

Le problème complet devient :

$$\begin{aligned} \min \sum_{k=1}^K u_k \\ \sum_{k=1}^K x_{ik} \geq d_i, \quad \forall i \in \{1, \dots, N\} \end{aligned} \tag{1.3}$$

$$\sum_{i=1}^N l_i x_{ik} \leq L_k u_k, \quad \forall k \in \{1, \dots, K\} \tag{1.4}$$

$$\begin{aligned} x_{ik} \in \mathbb{N}, \quad \forall i \in \{1, \dots, N\}, \forall k \in \{1, \dots, K\} \\ u_k \in \{0, 1\}, \quad \forall k \in \{1, \dots, K\} \end{aligned}$$

Ce modèle contient  $K(1 + N)$  variables entières.

### Décomposition de Dantzig-Wolfe

Nous appliquons maintenant la décomposition de la section précédente. Nous allons prendre pour partie  $A$  les contraintes (1.3) et pour  $k$  parties  $B_k$  les contraintes (1.4).

Pour chaque  $k \in \{1 \dots K\}$ , l'ensemble  $B_k$  défini par :

$$\begin{aligned} \sum_{i=1}^N l_i x_{ik} \leq L_k u_k, \\ x_{ik} \in \mathbb{N}, \quad \forall i \in \{1, \dots, N\} \end{aligned}$$

peut être représenté comme combinaison linéaire de ses points extrêmes ( $x_k^1, \dots, x_k^Q$ ). Soient  $\lambda_k^1, \dots, \lambda_k^Q$  les coefficients de  $x$  dans cette décomposition. Nous avons alors les équations de base :

$$x_{ik} = \sum_{q=1}^Q \lambda_k^q x_{ik}^q$$

$$u_k = \sum_{q=1}^Q \lambda_k^q$$

Le problème maître est alors :

$$\min \sum_{k=1}^K \sum_{q=1}^Q \lambda_k^q$$

$$\sum_{k=1}^K \sum_{q=1}^Q x_{ik}^q \lambda_k^q \geq d_i, \quad \forall i \in \{1, \dots, N\}$$

$$\sum_{q=1}^Q \lambda_k^q = 1, \quad \forall k \in \{1, \dots, K\}$$

$$\lambda_k^q \in \{0, 1\}, \quad \forall k \in \{1, \dots, K\}, \forall q \in \{1, \dots, Q\}$$

Le sous-problème pour  $B_k$  étant :

$$\min (1 - \pi)x - \pi_0$$

$$\sum_{i=1}^N l_i x_i \leq L_k$$

### Modèle agrégé

En prenant en plus le cas particulier où toutes les barres sont identiques (i.e.  $L_k = L$ ), nous pouvons identifier tous les sous-problèmes et obtenons :

$$\min \sum_{q=1}^Q \lambda^q$$

$$\sum_{q=1}^Q x_i^q \lambda^q \geq d_i, \quad \forall i \in \{1, \dots, N\}$$

$$\sum_{q=1}^Q \lambda^q = 1$$

Et le sous-problème :

$$\begin{aligned} \min(1 - \pi)x - \pi_0 \\ \sum_{i=1}^N l_i x_i \leq L \end{aligned}$$

Le problème maître ne contient plus que  $N$  contraintes, mais le nombre de variables est devenu potentiellement énorme. En effet, en s'approximant au cas où  $L$  est de l'ordre de 100, les  $l_i$  de l'ordre de 10 et  $N$  de l'ordre de 10, nous pouvons estimer le nombre de patrons (et donc de variables dans  $\Omega$ ) à  $\frac{100}{10}^{10} = 10^{10}$ . Nous utilisons alors des méthodes de génération de colonnes en utilisant un problème maître restreint.

### Génération de colonnes simple

Nous entendons par *génération de colonnes simple* la procédure pour résoudre un problème de génération de colonnes à variables entières introduite initialement dans [GG61]. Elle consiste à résoudre le problème restreint relâché (relaxation linéaire de la formulation de DW) et à lui ajouter les colonnes de coût réduit favorable, tant que ceci est possible. Nous obtenons alors la solution optimale au problème complet relâché. Le problème surgit quand nous voulons obtenir des solutions entières (quand les  $\lambda$  doivent être entiers). Pour cela, il est possible d'effectuer une recherche de type PLNE uniquement sur les colonnes ajoutées explicitement au problème maître. Il est alors important de garder à l'esprit que la solution entière optimale sur ces colonnes n'est pas nécessairement optimale sur le problème complet. Les colonnes générées lors de cette première phase peuvent même être insuffisantes pour obtenir une solution entière. Afin d'assurer l'optimalité de la solution, des méthodes plus complexes devront être mises en œuvre, telle la méthode de *Branch-and-Price* qui sera présentée dans la section 2.3.

### 1.3.2 Tournées de véhicules

Dans cette section, nous présentons un exemple montrant la manière dont s'effectue la décomposition et quels sont les deux problèmes obtenus lorsqu'il s'agit d'un cas où le sous-problème est un problème de plus court chemin.

Par la suite, cette phase de décomposition est mise de côté. En effet, la formulation décomposée dans une grande majorité des cas peut être vue comme aussi naturelle que la formulation d'origine. Nous utiliserons donc directement la formulation en colonnes sans reprendre la méthode de décomposition.

Nous illustrons cette décomposition sur un problème de tournées de véhicules. Cependant, afin de simplifier cette illustration nous nous contenterons d'un pro-

blème de tournées sans fenêtres de temps. Le même type de décomposition est effectué pour le problème avec fenêtres de temps dans [CDD<sup>+</sup>99].

Comme nous l'avons fait précédemment pour le problème de découpe, nous commencerons par décrire formellement le problème, puis donnerons un modèle de PLNE, lui appliquerons la décomposition de Dantzig-Wolfe et finalement, nous simplifierons le modèle décomposé en effectuant une hypothèse sur l'équivalence des véhicules.

### Le problème de VRP

Le problème de VRP (acronyme anglophone de *Vehicule Routing Problem*) peut se formuler de la manière suivante. Soient  $K$  véhicules identiques de capacité  $C_k$  initialement présents en un point particulier appelé dépôt. Soient  $N$  visites auxquelles doivent être déposées en une seule fois des quantités de matière  $d_i$ . Soit  $A$  un ensemble d'arcs autorisés entre ces  $N + 1$  sites. Soit un distancier  $d$  définissant une distance pour chaque arc  $(i, j) \in A$ . Ce distancier est identique pour tous les véhicules. Les routes effectuées par les véhicules devant commencer et finir au dépôt, nous ajouterons deux visites fictives 0 et  $N + 1$ . Le problème consiste à trouver les routes suivies par les  $K$  véhicules afin de minimiser la distance totale qu'ils effectuent.

### Modèle PLNE

Le modèle de programmation linéaire le plus fréquemment utilisé pour ce type de problème consiste à utiliser des variables  $x_{ijk}$  indiquant si le véhicule  $k$  effectue le trajet du nœud  $i$  au nœud  $j$ . Les contraintes devant être prises en compte correspondent alors à assurer que les variables représentent bien un chemin valide et unique pour chaque véhicule, et que les capacités des véhicules soient respectées.

Un modèle de PLNE est :

$$\min \sum_{k=1}^K \sum_{(i,j) \in A} d(i,j)x_{ijk} \quad (1.5)$$

$$\sum_{j \in \delta^+(0)} x_{0jk} = 1, \quad \forall k \in \{1, \dots, K\} \quad (1.6)$$

$$\sum_{i \in \delta^-(j)} x_{ijk} - \sum_{i \in \delta^+(j)} x_{jik} = 0, \quad \forall k \in \{1, \dots, K\}, \forall j \in \{1, \dots, N\} \quad (1.7)$$

$$\sum_{i \in \delta^-(N+1)} x_{i,N+1,k} = 1, \quad \forall k \in \{1, \dots, K\} \quad (1.8)$$

$$\sum_{i \in S} \sum_{j \in S, j \neq i} x_{ijk} \leq |S| - 1 \quad \forall k \in \{1, \dots, K\}, \forall S \subset X, 1 < |S| < N \quad (1.9)$$

$$\sum_{k=1}^K \sum_{j \in \delta^+(i)} x_{ijk} = 1, \quad \forall i \in \{1, \dots, N\} \quad (1.10)$$

$$\sum_{i=1}^N d_i \sum_{j \in \delta^+(i)} x_{ijk} \leq C_k, \quad \forall k \in \{1, \dots, K\} \quad (1.11)$$

$$x_{ijk} \in \{0, 1\}, \quad \forall k \in \{1, \dots, K\}, \forall (i, j) \in A \quad (1.12)$$

Les contraintes (1.6), (1.7) et (1.8) caractérisent le flux suivi par le véhicule  $k$ , la contrainte (1.10) limite l'affectation d'une visite à un unique véhicule, et la contrainte (1.11) limite la capacité d'un véhicule  $k$ . Les contraintes (1.9) permettent d'éliminer les sous-tours. Enfin, l'objectif (1.5) reflète le fait que le coût total est donné par la somme des coûts des arcs traversés.

Notons que les contraintes d'élimination des sous tours sont généralement ajoutées dynamiquement lorsque ce type de modèle est utilisé.

## Décomposition

De même que nous l'avons fait dans la section précédente pour le problème de découpe à une dimension, nous allons appliquer la décomposition de Dantzig-Wolfe à ce problème.

Ici, plusieurs ensembles de contraintes  $B_k$  sont considérés, un pour chaque véhicule. Chaque matrice  $B_k$  correspond aux contraintes (1.6), (1.7), (1.8), (1.9) et (1.11). La contrainte (1.10) correspond à la matrice  $A$ . Le polytope défini par  $B_k$  a pour points extrêmes l'ensemble  $\Omega_k$  des chemins valides pour le véhicule  $k$ . Nous pouvons donc écrire une solution comme une combinaison linéaire de ces chemins en utilisant les coefficients  $x_{ijk}^p$ , valant 1 si l'arc  $(i, j)$  appartient au

chemin du point extrême  $\lambda_k^p$ , et 0 sinon. Pour chaque  $k \in \{1, \dots, K\}$ , une solution  $x_{ijk}$  peut se réécrire comme une combinaison linéaire de chemins :

$$\begin{aligned} x_{ijk} &= \sum_{p \in \Omega_k} x_{ijk}^p \lambda_k^p, & \forall (i, j) \in A, \\ \sum_{p \in \Omega_k} \lambda_k^p &= 1 \\ \lambda_k^p &\geq 0, & \forall p \in \Omega_k \end{aligned}$$

Nous pouvons alors définir  $c_k^p$  comme le coût de la route  $p$  pour le véhicule  $k$ . Aussi, les entiers positifs  $a_{ik}^p$  indiquent le nombre de fois où la visite  $i$  est effectuée par le véhicule  $k$  sur la route  $p$ . Pour les routes de  $\Omega_k$ , nous avons  $a_{ik}^p \in \{0, 1\}$ . Pour chaque  $k \in \{1, \dots, K\}$ , et chaque route  $p \in \Omega_k$ , nous avons alors :

$$\begin{aligned} c_k^p &= \sum_{(i,j) \in A} d(i, j) x_{ijk}^p, \\ a_{ik}^p &= \sum_{j \in \delta^+(i)} x_{ijk}^p, & \forall i \in \{1, \dots, N\} \end{aligned}$$

En substituant ces expressions dans le modèle initial, nous obtenons le modèle décomposé :

$$\begin{aligned} \min & \sum_{k=1}^K \sum_{p \in \Omega_k} c_k^p \lambda_k^p \\ \sum_{k=1}^K \sum_{p \in \Omega_k} a_{ik}^p \lambda_k^p &= 1, & \forall i \in \{1, \dots, N\} \\ \sum_{p \in \Omega_k} \lambda_k^p &= 1, & \forall k \in \{1, \dots, K\} \\ \lambda_k^p &\geq 0, & \forall k \in \{1, \dots, K\}, \forall p \in \Omega_k \end{aligned}$$

La contrainte de capacité ainsi que les contraintes d'élimination des sous-tours se retrouvent intégralement dans le sous-problème de génération de nouvelles routes. Le sous-problème pour le véhicule  $k$  se formule alors ainsi :

$$\begin{aligned}
& \min \sum_{(i,j) \in A} d'(i,j)x_{ijk} \\
& \sum_{j \in \delta^+(0)} x_{0jk} = 1, \\
& \sum_{i \in \delta^-(j)} x_{ijk} - \sum_{i \in \delta^+(j)} x_{jik} = 0, \quad \forall j \in \{1, \dots, N\} \\
& \sum_{i \in \delta^-(N+1)} x_{i,N+1,k} = 1, \\
& \sum_{i \in S} \sum_{j \in S, j \neq i} x_{ijk} \leq |S| - 1 \quad \forall S \subset X, 1 < |S| < N \\
& \sum_{i=1}^N d_i \sum_{j \in \delta^+(i)} x_{ijk} \leq C_k, \\
& x_{ijk} \in \{0, 1\}, \quad \forall (i, j) \in A
\end{aligned}$$

Ce sous-problème peut être présenté plus simplement sous une forme équivalente. En effet, nous reconnaissons un problème de plus court chemin sans sous-tours du dépôt au dépôt passant par chaque visite au plus une fois et respectant la contrainte de capacité du véhicule. Ce problème n'est bien sûr pas résolu par la programmation linéaire mais avec un algorithme spécifique.

### Cas de véhicules identiques

Finalement, si les véhicules sont identiques et de même capacité  $C$ , nous pouvons regrouper certaines parties du modèle en posant :

$$\lambda^p = \sum_{k=1}^K \lambda_k^p, \forall p \in \Omega = \bigcup_{k=1}^K \Omega_k$$

et obtenir le problème maître :

$$\begin{aligned}
& \min \sum_{p \in \Omega} c^p \lambda^p \\
& \sum_{p \in \Omega} a_i^p \lambda^p = 1, \quad \forall i = 1, \dots, N \tag{1.13}
\end{aligned}$$

$$\sum_{p \in \Omega} \lambda^p = K, \tag{1.14}$$

$$\lambda^p \geq 0, \quad \forall p \in \Omega$$

Les  $K$  sous-problèmes s'unifient également en un unique sous-problème :

$$\begin{aligned}
\min \quad & \sum_{(i,j) \in A} d'(i,j)x_{ij} \\
& \sum_{j \in \delta^+(0)} x_{0j} = 1, \\
\sum_{i \in \delta^-(j)} x_{ij} - \sum_{i \in \delta^+(j)} x_{ji} &= 0, \quad \forall j \in \{1, \dots, N\} \\
& \sum_{i \in \delta^-(N+1)} x_{i,N+1} = 1, \\
\sum_{i \in S} \sum_{j \in S, j \neq i} x_{ij} &\leq |S| - 1 \quad \forall S \subset X, 1 < |S| < N \\
\sum_{i=1}^N d_i \sum_{j \in \delta^+(i)} x_{ij} &\leq C, \\
x_{ij} &\in \{0, 1\}, \quad \forall (i,j) \in A
\end{aligned}$$

### Forme du coût réduit

Si nous notons respectivement  $\pi_i$  et  $\pi_0$  les valeurs duales correspondant aux équations (1.13) et (1.14), le coût réduit pour un chemin  $p$  est donné par :

$$\begin{aligned}
cr(p) &= c(p) - \sum_{i=1}^N a_i^p \pi_i - \pi_0 \\
&= \sum_{(i,j) \in p} d(i,j) - \sum_{i=1}^N a_i^p \pi_i - \pi_0 \\
&= \sum_{(i,j) \in p} (d(i,j) - \pi_j) - \pi_0
\end{aligned}$$

Le sous-problème peut alors être défini comme le problème consistant à trouver le plus court chemin de la visite 0 à la visite  $N + 1$  en prenant comme fonction de coût le coût réduit :

$$\begin{aligned}
d'(i,j) = cr(i,j) &= d(i,j) - \pi_j, \forall j \in \{1, \dots, N\} \\
cr(i, N+1) &= -\pi_0
\end{aligned}$$

### 1.3.3 Problèmes de plus court chemin

Dans l'exemple précédent, le sous-problème apparaît sous forme d'un problème de plus court chemin. En effet, trouver la colonne de  $\Omega^* \setminus \Omega$  de meilleur coût réduit, correspond à trouver le chemin de 0 à  $N + 1$  dont le coût est défini comme la somme des coûts des arcs donnés par  $cr(i, j)$ , et dont la somme des poids des charges distribuées dans les visites est inférieure à la capacité du véhicule.

Si les véhicules n'avaient pas eu de capacité maximale (capacité supposée infinie), ce problème aurait été encore plus simple. Au contraire, si par exemple, le nombre de visites par véhicules avait été limité, le sous-problème serait alors un problème de plus court chemin de longueur limitée. Nous voyons donc que de nombreux problèmes de plus court chemin peuvent surgir, avec de nombreuses variantes, dans les sous-problèmes.

Les problèmes de plus court chemin dans un graphe ont été très étudiés. Il existe de très nombreuses types de problèmes auxquels correspondent de nombreux algorithmes aux complexités variées. Une présentation complète de ces problèmes et des algorithmes correspondant est donnée dans [MG].

## 1.4 Applications

Nous avons vu que la décomposition de Dantzig-Wolfe, à la base des techniques de génération de colonnes permet de donner une forme plus pratique à certains problèmes. Nous avons illustré cette méthode à l'aide de deux exemples académiques simples. Nous allons maintenant présenter des exemples concrets pour lesquels ces méthodes peuvent être utiles. Il ne s'agit pas ici de donner des présentations formelles pour ces problèmes, mais de définir les caractéristiques communes qui font que les techniques de génération de colonnes peuvent être envisagées, puis de donner quelques exemples donnant une idée de problèmes concrets qui correspondent à ces caractéristiques. Des modèles seront donnés pour ces problèmes dans la partie III. Nous parlerons principalement de trois grandes familles de problèmes. Nous présenterons nos apports en utilisant une application appartenant à chacune de ces familles.

Dans la pratique, il peut être intéressant d'utiliser des méthodes de génération de colonnes pour un problème quand il remplit une ou plusieurs des conditions suivantes :

- décomposition en contraintes locales et globales (correspondant aux matrices  $A$  et  $B$  que nous avons considérées dans la décomposition),
- contraintes globales linéaires (celles de  $A$ ),
- variables/colonnes du problème global correspondent à des solutions d'un problème contenant les contraintes locales,
- contraintes locales complexes (celles de  $B$ ),

- beaucoup de colonnes (dans le cas contraire, on peut utiliser la décomposition et résoudre le PLNE complet).

Ces conditions ne sont bien sûr pas toutes nécessaires. Par exemple, si le nombre de colonnes est vraiment très grand, l'utilisation de méthodes de génération de colonnes peut être envisagée même si les contraintes locales sont simples à traiter.

Dans cette section, nous présentons trois familles de problèmes qui nous semblent intéressants tant par la diversité du domaine d'application que par l'étendue possible de leurs extensions. Cette liste n'est bien sûr pas exhaustive. A chacune des familles, nous associerons par ailleurs un problème de base type qui sera celui sur lequel nous développerons nos contributions.

### 1.4.1 Tournées de véhicules et variantes

Les problèmes de tournées de véhicules ont été très étudiés dans des études variées utilisant des méthodes de résolution elles-aussi très variées. Une synthèse académique de ces problèmes peut être trouvée dans le problème de tournées de véhicules avec fenêtre de temps (VRPTW : *Vehicle Routing Problem with Time Windows*) dont une introduction complète est donnée dans [SD88].

#### Problème de base

Présentons brièvement la famille des problèmes de tournée de véhicules avec fenêtre de temps. Un ensemble de nœuds étant donné, parmi lesquels un nœud particulier correspond à ce que nous appellerons le *dépôt*, il s'agit de trouver un ensemble de tournées pouvant être effectuées par des véhicules sortant du dépôt, effectuant un certain nombre de visites à d'autres nœuds, puis retournant au dépôt. Chacune de ces tournées doit individuellement vérifier un certain nombre de contraintes :

- à chaque visite correspond une certaine quantité de produits à livrer, la somme des quantités correspondant aux visites effectuées doit être inférieure à une capacité maximale donnée pour le véhicule,
- à chaque paire de nœuds correspond une distance représentant le temps nécessaire pour parcourir cet arc, l'accumulation de distance du dépôt à chaque visite doit être comprise dans une fenêtre de valeur donnée. Cette contrainte représente les horaires d'ouverture des clients présents à chaque nœud.

Globalement, il s'agit :

- d'assurer que chacune des visites sera effectuée, une et une seule fois, par l'une des tournées effectuées par l'un des véhicules,

- de choisir, parmi tous les ensembles de tournées réalisables, celui assurant une distance totale minimale.

### Extensions possibles

A partir de ce problème simplifié, de nombreuses extensions sont possibles : minimiser non seulement la distance totale parcourue, mais aussi le nombre de véhicules utilisés, autoriser l'utilisation de plusieurs dépôts, avoir des formats de fenêtres de temps plus complexes, avoir d'autres types de contraintes s'appliquant sur les tournées.

### Méthodes de résolution

Il existe deux grandes familles de méthodes de résolution utilisées pour ce type de problème : la recherche locale et la génération de colonnes.

La recherche locale [RT95, CLM01, HG99, GTA99] permet d'obtenir des solutions de bonne qualité dans un temps limité. Étant donnée une solution ou un ensemble de solutions, des opérateurs définissant un *voisinage* sont utilisés pour modifier de proche en proche cette solution. Diverses méta-heuristiques (recuit simulé, recherche tabou, etc.) sont couramment utilisées pour guider la recherche. L'avantage indéniable de ces méthodes est qu'elles permettent d'obtenir rapidement des solutions de qualité. Elles présentent par contre deux inconvénients. Tout d'abord, elles ne fournissent pas de borne inférieure. L'utilisateur ne dispose donc d'aucun majorant de la différence entre une solution optimale (inconnue) et une solution obtenue en un temps donné. Il lui est dès lors difficile de décider s'il se satisfait de la meilleure solution disponible ou s'il doit allouer plus de temps à la résolution. Par ailleurs, l'intégration de nouvelles contraintes à un algorithme de résolution par recherche locale n'est pas toujours facile : les bonnes propriétés des voisinages utilisés pouvant éventuellement être détruites par l'addition de nouveaux types de contraintes.

Ce dernier inconvénient a amené différents chercheurs à intégrer des techniques de programmation par contraintes et des techniques de recherche locale [DFS<sup>+</sup>00, RGP99, Sha98, KPS00, BH01, CLLR01]. Une telle hybridation a permis de mieux répondre à l'existence de contraintes additionnelles et de réaliser des applications qui sinon n'auraient pas vu le jour. Néanmoins, la définition du voisinage reste au cœur de la méthode et il peut être difficile d'évaluer la qualité effective des solutions obtenues.

Parmi les méthodes exactes, la génération de colonnes a été appliquée avec succès à une variante importante du problème de routage de véhicules, le VRPTW : [CR99, Lar99, KLM01].

La méthode de génération de colonnes présente deux avantages principaux : elle fournit une borne inférieure, qui se trouve affinée durant la recherche ; elle

permet de traiter les contraintes internes à une tournée au sein du sous-problème de génération de tournées et non au sein du problème global, ce qui facilite l'intégration de nombreuses contraintes additionnelles. Par contre, elle a la réputation de ne pas permettre d'obtenir *rapidement* des bonnes solutions.

Notons finalement, que les deux méthodes peuvent être hybridées dans le but de profiter des avantages de chacune. C'est ce que nous avons proposé dans [CDP02].

### Modèles de génération de colonnes

Les modèles de génération de colonnes sont en effet très adaptés à ce type de problèmes comme nous avons pu le voir en effectuant la décomposition dans la section 1.3.2. Au niveau du problème maître, seules les contraintes de couvertures des visites seront prises en compte. Les contraintes de validité des tournées sont pour leur part toutes intégrées au sous-problème. Leur complexité n'est limitée que par le type d'algorithme utilisé pour résoudre le sous-problème.

### 1.4.2 Conception de réseau

Les problèmes de conception de réseau ont également été le sujet de nombreuses recherches et publications. Ils sont l'extension naturelle des problèmes de flux. Ceux-ci peuvent porter sur le flux d'une ou plusieurs commodités, ces flux devant être entiers ou non, et passant dans un graphe avec des arcs de capacité booléenne, discrète ou infinie. Diverses publications offrent une introduction au problème de conception de réseau : [Gun98], [CG95] ou encore [MM93].

#### Problème de base

Le problème de conception de réseau recouvre une grande famille de problèmes consistant à dimensionner un réseau de télécommunications. Étant donné un graphe, il s'agit de décider sur lesquels de ses arcs attribuer des capacités de communication permettant d'acheminer au moindre coût un ensemble de communications donné.

Nous avons utilisé les nombreuses instances du problème de conception de réseau utilisées dans le cadre du projet ROCOCO qui sont disponibles publiquement. Une description complète du problème de référence, de ses variantes, et des jeux de données peut être trouvée dans [BCP<sup>+</sup>02].

#### Extensions possibles

De nombreuses contraintes annexes peuvent être prises en compte. Le benchmark proposé dans [BCP<sup>+</sup>02] propose d'ailleurs de telles contraintes. En effet, un

ensemble de 6 types de contraintes additionnelles est proposé. Il s'agit donc de trouver des algorithmes permettant de donner de bons résultats sur l'ensemble des  $2^6 = 64$  configurations de contraintes. Parmi ces contraintes additionnelles, citons :

- la sécurisation de certaines demandes qui ne peuvent emprunter certains nœuds ou arcs considérés à risque,
- l'interdiction de combiner les différents types de capacité sur chaque arc,
- la symétrie des chemins suivis par des demandes symétriques,
- la limitation du nombre de bonds effectués par certaines demandes,
- la limitation du nombre de liens attachés à certains nœuds,
- la limitation du trafic passant par certains nœuds.

### Méthodes de résolution

La publication de ce *benchmark* a permis la comparaison de nombreuses méthodes de résolution. Citons la programmation par contrainte (PPC), ainsi que sa parallélisation, introduite dans l'article d'origine [BCP<sup>+</sup>02]. Mais d'autres méthodes ont été utilisées, parmi lesquelles la programmation linéaire en nombres entiers, la recherche locale et bien sûr la génération de colonnes.

Au delà des travaux réalisés dans le cadre du projet ROCOCO, de nombreuses autres publications traitent du problème de conception de réseau. Citons par exemple [BG96], [Gun98], [CG95] et [GKM99].

### Modèles de génération de colonnes

Le modèle de génération de colonnes pour ce problème fait intervenir deux types différents de colonnes : les chemins suivis par les demandes, et les liens ouverts sur certains arcs. Parmi les contraintes du problème maître, la principale contrainte indique que pour chaque arc, le flux de demande passant par cet arc doit être inférieur à la capacité des liens ouverts sur cet arc. Contrairement à d'autres modèles de génération de colonnes, certaines contraintes additionnelles auront pour résultats non seulement une modification du sous-problème mais également l'ajout de certaines contraintes dans le problème maître.

#### 1.4.3 Planification de ressources

Par planification de ressources, nous entendons attribuer à des activités dont les dates et conditions d'exécution sont déjà fixées, des ressources qui sont en nombre limité. L'industrie du transport aérien est confrontée à de nombreux problèmes différents appartenant à cette famille. Nous utiliserons comme problème représentatif de cette famille le problème de la génération de *pairings*. Une

bonne introduction aux problèmes de planification de ressources dans le transport aérien est donnée dans [Yu98] et [VBJN95].

### Différences avec l’ordonnancement

Dans les problèmes de cette famille, les dates d’exécution des activités sont fixées à l’avance. Aucun ordonnancement ni séquençement n’a besoin d’être réalisé. A chacune des activités doit simplement être affectée une ressource (une personne, une machine, etc) de telle manière que la suite des activités réalisées par une même ressource respecte un ensemble de contraintes (contraintes physiques, conventions collectives, réglementations, etc) propre à chaque type de ressource.

### Problème de base

Nous présentons ici brièvement le problème de génération de *pairings*. Un ensemble de vols à couvrir étant donné, nous appellerons un *pairing* une succession de vols qui sera effectuée par le même membre d’équipage. Ce pairing est en général décomposable en plusieurs *activités aériennes*, séries de vols sans repos. Le problème consiste alors à trouver un ensemble de *pairings* respectant certaines réglementations et conventions collectives, couvrant l’ensemble des vols, et ayant un coût total minimal. Les réglementations s’appliquant sur un pairing sont assez variables. Parmi celles que nous avons prises en compte, citons :

- un repos minimum entre deux activités aériennes de durée supérieure à une fonction de l’activité précédente,
- un nombre de jours maximal sur la durée total du pairing,
- la succession correcte à priori des aéroports de fin et de début de vols successifs, celle-ci pouvant ne pas être respectée dans certains cas précis, et dans tous les cas uniquement entre deux activités aériennes différentes,
- le premier et le dernier vols doivent respectivement commencer et finir dans un même aéroport, appelée *base*.

### Extensions possibles

Les extensions possibles étant infinies, il n’existe aucun benchmark de référence sur ce problème, chaque groupe de recherche travaillant sur une variante différente du problème. La principale variante sur laquelle nous avons travaillé porte sur le système de coûts à utiliser, celui-ci pouvant prendre en compte, en plus du nombre de jours, le nombre d’heures d’activités aériennes, le nombre de nuits passées en dehors de la base, etc. Il est même fréquent d’utiliser une fonction plus complexe sur l’ensemble de ces coûts, comme par exemple le maximum. Les réglementations changent pour chaque pays et les conventions pour chaque compagnie aérienne.

## Méthodes de résolution

A notre connaissance, ce sont principalement des méthodes de génération de colonnes qui ont été utilisées dans ce domaine. Il s'agit d'ailleurs du domaine où les recherches sur la génération de colonnes ont été les plus intenses et ont amené le plus grand nombre d'innovations.

## Modèles de génération de colonnes

Celui-ci est en général assez simple, une contrainte maître existant pour chacun des vols à couvrir. Toutes les réglementations et conventions sont en général prises en compte uniquement au niveau du sous-problème qui lui peut avoir une forme très complexe.

### 1.4.4 Caractéristiques communes de ces problèmes

Les trois familles de problèmes introduites précédemment peuvent toutes être traitées avec des méthodes de génération de colonnes. De plus, contrairement aux problèmes de découpe qui historiquement ont été à la base du développement des méthodes de génération de colonnes, ils ont tous en commun que les colonnes sont des chemins dans des graphes :

- une tournée d'un véhicule traversant certaines visites,
- un chemin utilisé pour une communication dans un réseau,
- une suite ordonnée d'activités effectuées par une même ressource.

Au delà de cette simple similitude sur la forme que peut prendre la décomposition du problème, bien d'autres aspects peuvent être mis en parallèle.

Les contraintes locales prendront également des formes similaires :

- succession de visites, ou d'activités contraintes, correspondant à l'existence ou non d'arcs dans le graphe de recherche des chemins,
- forme du coût (et donc du coût réduit)
- et donc, à terme, de la méthode de résolution du sous problème, qui correspond logiquement à un problème de plus court chemin contraint.

Les contraintes globales prendront des formes similaires :

- nombre maximal de véhicules ou de ressources d'un certain type, ou quantité maximale de trafic passant par un noeud du réseau de communication, correspondant à une ligne avec majorant dans le problème maître,
- quantité minimale à livrer à un client, ou nombre minimal de ressources nécessaires pour exécuter une activité, correspondant à une ligne avec minorant dans le problème maître.

Nous voyons donc que de nombreuses similitudes existent entre ces problèmes tant au niveau du problème maître, que du sous-problème et que des interactions entre ceux-ci. Ces similitudes auront pour résultat une importante source

de factorisation dans la modélisation des problèmes ainsi que dans les méthodes de résolution.

Bien que la littérature se propose en général de résoudre un type de problème précis avec une méthode de génération de colonnes, seul [DDS01] présente un effort pour généraliser des résultats obtenus sur différents problèmes.

Cette thèse a pour sujet la réutilisation de résultats entre tous ces problèmes. Pour cela, une étape nécessaire consiste à reconnaître les principales similitudes. Après avoir, dans le prochain chapitre, présenté un état de l'art de toutes les propositions ayant été faites pour certains de ces problèmes, nous proposerons dans les chapitres 3 et 4 un formalisme générique pour la modélisation et la résolution de l'ensemble de ces problèmes.



# Chapitre 2

## État de l'art

Dans le chapitre précédent, nous avons essayé d'introduire de manière didactique les méthodes de génération de colonnes avec sous-problème de plus court chemin et de situer les problèmes qu'elles permettent de résoudre.

Dans ce chapitre, nous présentons de façon plus systématique les nombreuses améliorations qui ont été proposées durant ces dernières années. Chaque section de ce chapitre est dédiée à une partie particulière de la méthode : la modélisation (2.1), la génération de colonnes simple (2.2), la méthode de *Branch-and-Price* (2.3), les problèmes de plus court chemin (2.4), les méthodes d'accélération (2.5) (du problème maître, du sous-problème ou mettant en jeu les deux), et finalement la méthode du *Branch-and-Price-and-Cut* (2.6).

### 2.1 Modèles

Dans la littérature, les modèles de génération de colonnes sont présentés de deux manières différentes qui nous semblent être liées à l'origine du ou des auteur(s).

D'une part, les auteurs originaires du domaine de la programmation linéaire présentent souvent un premier modèle de PLNE à partir duquel est ensuite obtenu le modèle décomposé. La décomposition est parfois présentée. L'attention est alors portée sur le problème maître dont le modèle est donné en utilisant le formalisme habituel de la programmation linéaire (tel que nous l'avons fait dans le chapitre précédent). Le sous-problème n'est étudié en profondeur que quand il s'agit d'un problème typique comme un problème de sac-à-dos ou un problème de plus court chemin. Ces études proposent en général des améliorations relatives au problème maître qui sont souvent des applications de techniques provenant de la programmation linéaire.

D'autre part, les auteurs originaires de la programmation par contraintes ou de la recherche locale portent leur attention sur le sous-problème. Le problème

maître est donné brièvement et la décomposition n'est quasiment jamais abordée. Un modèle pour le sous-problème est souvent énoncé directement. Ces études abordent en général des situations où le sous-problème est difficile à résoudre.

Finalement, les interactions entre les deux problèmes ainsi que la procédure de résolution globale ne sont que rarement détaillés.

## 2.2 Génération de colonnes simple

Comme nous l'avons souligné en introduisant la méthode de décomposition, le modèle décomposé contient énormément de variables correspondant à des colonnes devant être entières. La technique consistant à ne prendre en compte qu'un problème maître restreint à un sous ensemble de variables et à simplement appliquer la méthode révisée du simplexe ne peut être suffisante que pour des problèmes où les variables sont continues. De plus, la condition sur le coût réduit des variables hors problème maître restreint permettant d'affirmer l'optimalité de la solution courante n'est valable que dans le cas d'un problème en variables continues. En génération de colonnes, comme en PLNE, les contraintes d'intégralité sont alors *relâchées*. Un *problème relâché* est obtenu sur lequel la méthode génération de colonnes est appliquée.

### 2.2.1 Solution du problème relâché

La solution optimale obtenue est alors optimale pour le problème relâché. Il est possible que toutes les variables de type entier dans le problème d'origine aient des valeurs entières dans cette solution. Celle ci sera alors optimale non seulement pour le problème relâché, mais aussi pour le problème entier. En effet, s'il existait une solution entière meilleure, elle serait aussi meilleure pour le problème relâché, ce qui contredirait l'optimalité de la première.

D'autre part, ce qui est malheureusement plus fréquent, la solution optimale relâchée peut contenir des variables, entières dans le problème d'origine, mais ayant des valeurs non-entières dans la solution optimale relâchée. Cette solution n'est donc pas entière. Elle ne constitue que la solution optimale relâchée, et sa valeur est d'autre part une borne inférieure du coût optimal du problème entier.

### 2.2.2 Solutions entières

Pendant longtemps, l'obtention d'une solution entière s'est réduite à résoudre le PLNE obtenu en prenant en compte toutes les colonnes (mais seulement elles) qui ont été générées durant la résolution du problème relâché. Nous nous référons à cette méthode sous la terminologie de *Génération de Colonnes Simple*. Cette méthode peut s'avérer suffisante dans certains cas sur lesquels nous reviendrons

à la fin de cette section. Cependant, il faut tout d'abord noter que la solution optimale alors obtenue en résolvant ce PLNE n'est pas garantie comme étant optimale pour le problème entier complet. De plus, il est tout à fait possible qu'aucune solution entière n'existe sur ces colonnes alors qu'une solution existe pour le problème entier complet d'origine. Ce dernier point n'est finalement qu'un cas particulier du précédent.

Illustrons cette possibilité sur un exemple. Reprenons le problème de tournées de véhicules que nous avons utilisé pour illustrer la méthode de décomposition avec sous-problème de plus court chemin dans la section 1.3.2. Imaginons que seules deux visites, 1 et 2, existent, chacune avec une quantité de matière à déposer de 5 unités et la capacité du véhicule étant limitée à 20 unités. Les distances utilisées sont données dans le tableau 2.1 (nous notons  $d$  les deux visites identiques correspondant au dépôt).

	d	1	2
d		100	100
1	100		1
2	100	1	

TAB. 2.1 – Exemple de distancier

Imaginons que le processus de génération de colonnes du problème relâché n'a consisté qu'à la génération d'une unique colonne fournissant une solution optimale :

$$(d - 1 - 2 - 1 - 2 - d)$$

Le coût de cette route est de  $100 + 1 + 1 + 1 + 100 = 203$ . La figure 2.1 montre cette solution. Une solution au problème relâché consiste à prendre cette route  $\frac{1}{2}$  fois, pour un coût total de  $\frac{203}{2}$ . Il est possible de vérifier que cette solution est optimale pour le problème complet relâché. Quel que soit le nombre de colonnes générées et l'ordre dans lequel celles-ci sont ajoutées, cette solution optimale sera finalement atteinte. Il est également facile de voir que le PLNE ne comprenant que cette colonne ne peut fournir aucune solution entière ne couvrant chacune des visites une et une seule fois. Nous sommes donc dans une situation où une méthode autre que la *génération de colonnes simple* devra être utilisée. Un type de méthode permettant de résoudre ce problème sera présenté dans la prochaine section.

Revenons cependant sur ces cas où la solution du problème optimal n'est pas entière. Au delà des situations où la génération de colonnes simple ne fournit aucune solution entière, dans une majorité de situations où le nombre de colonnes

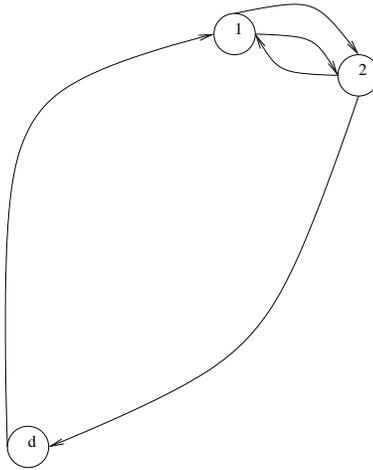


FIG. 2.1 – Solution de l'exemple de VRP

généérées dans la résolution du problème maître relâché a été grand, une solution entière peut être trouvée ayant un écart avec la solution relâchée ne permettant pas de déduire l'optimalité. Dans certains cas, même si les deux solutions diffèrent, l'optimalité de la solution entière peut être déduite. C'est le cas par exemple si la solution optimale relâchée a un coût non entier, que la solution entière a pour coût le premier entier supérieur, et que l'on sait que la solution entière doit avoir un coût entier.

### 2.3 Branch-and-Price

Comme nous venons de le montrer sur un exemple concret, la *génération de colonnes simple* peut ne pas fournir de solution au problème en variables en nombres entiers. En effet, comme cela est expliqué dans [BJN<sup>+</sup>98], la relaxation n'est pas nécessairement entière et l'application d'une méthode de *branch-and-bound* standard sur le problème maître restreint avec les colonnes existantes ne garantit pas l'optimalité (ni même l'existence) d'une solution. Une méthode arborescente doit alors être utilisée. En effet, après le branchement, le problème est modifié et il est possible qu'il existe une colonne qui ne soit pas présente dans le problème maître restreint mais qui ait un coût réduit favorable. Ainsi, il est donc nécessaire d'essayer de générer des colonnes après avoir effectué le branchement. Cette méthode est présentée dans [BJN<sup>+</sup>98] sous le nom de Branch-and-Price.

### 2.3.1 Dualité avec la méthode de *Branch-and-Cut*

Le *Branch-and-Price* pur est tout simplement l'algorithme symétrique de la méthode de *Branch-and-Cut* beaucoup plus répandue. Cette dernière est utilisée fréquemment pour les problèmes contenant un nombre trop élevé de contraintes pour être directement traitées. Certaines des contraintes sont donc exclues du problème linéaire initial. La méthode du simplexe est alors appliquée à la relaxation de ce problème *restreint*. Avant d'effectuer un éventuel branchement, les contraintes exclues du problème restreint sont parcourues et il est vérifié qu'elles ne sont pas *violées* par la solution primale courante. Au moins une des contraintes violées est ajoutée au problème restreint. Quand aucune des contraintes non présentes dans le problème restreint n'est violée par la solution primale courante, la solution relâchée courante offre alors une borne inférieure, voire une solution si les contraintes d'intégralité sont respectées. Si cette dernière condition n'est pas respectée, il est alors commun de brancher. Après le branchement, les contraintes externes devront être parcourues de nouveau. En effet, la solution primale du problème restreint relâché étant modifiée, la nouvelle solution peut violer l'une des contraintes non ajoutées.

La méthode de *Branch-and-Price* est dans son principe identique. Un tableau d'*équivalence* pour passer d'un algorithme à l'autre, pourrait être celui du tableau 2.2.

<i>Branch-and-Cut</i>	<i>Branch-and-Price</i>
Contrainte primale	Variable primale
Variable duale	Contrainte duale
Solution primale violée par une contrainte primale	Solution duale violée par une variable primale

TAB. 2.2 – Équivalences *Branch-and-Price* et *Branch-and-Cut*

Les variables du problème primal sont des contraintes pour le problème dual. Donc, en regardant le problème par sa formulation duale, toutes les variables exclues du problème maître restreint primal sont en fait des contraintes exclues du problème maître restreint dual. Regarder si ces contraintes sont violées est équivalent à regarder si leur variable d'écart est non nulle, et donc à regarder le coût réduit de la variable primale correspondante.

Pour la même raison qu'ajouter une coupe dans le problème primal peut changer la solution primale (la rendre moins bonne) et rendre nécessaire de regarder si d'autres contraintes doivent entrer dans le problème restreint, l'ajout d'une variable, i.e. d'une coupe dans le problème dual, modifie la solution duale et rend donc nécessaire de parcourir à nouveau les coûts réduits des variables non

ajoutées.

Appliquer la méthode de *Branch-and-Price* sur un problème linéaire est donc équivalent à appliquer la méthode de *Branch-and-Cut* sur son problème linéaire dual.

### 2.3.2 Règles de branchement

En pratique, l'utilisation de coupes quelconques pour brancher dans un problème de PLNE n'est pas répandue. Le branchement consiste souvent à simplement modifier les bornes d'une variable entière dont la valeur dans la relaxation est fractionnelle :

$$x \leq \lfloor x^* \rfloor \text{ ou } x \geq \lceil x^* \rceil$$

Ceci peut être considéré comme un cas particulier de la situation générale considérant le branchement comme une séparation du problème en une partition en plusieurs problèmes plus contraints. Dans chacun des problèmes est ajoutée au moins une coupe éliminant la solution non fractionnelle courante :

$$\sum_i a_i x_i \leq n \text{ ou } \sum_i a_i x_i \geq n + 1$$

Cette modification du problème par ajout d'une coupe rend nécessaire de répéter le test des contraintes (cas du *Branch-and-Cut*) ou des variables (cas du *Branch-and-Price*) non présentes dans le problème restreint.

La nécessité de tester de nouveau les contraintes hors problème restreint peut paraître plus simple à comprendre et à réaliser. En effet, l'ajout de la coupe de branchement modifie la solution optimale et cette nouvelle solution peut ne pas être valide pour une coupe hors problème restreint qui n'était pas violée par la solution précédente.

La situation du *Branch-and-Price* est en fait identique si l'on considère le problème dual. L'ajout de la coupe de branchement correspond à l'ajout d'une variable dans le problème dual. Cet ajout peut modifier la solution optimale relâchée (dans ce cas l'améliorer). Cette nouvelle solution peut violer une contrainte du problème dual qui n'était pas violée par la solution précédente. Cette contrainte duale correspond en fait à une variable primale. Regarder si cette contrainte duale est violée correspond également à regarder si la variable primale associée a un coût réduit favorable.

Si toutes les variables exclues du problème restreint sont connues, il est alors possible de tester leur coût réduit explicitement. Ceci n'est en général pas le cas, un des intérêts de la génération de colonnes étant justement de ne pas avoir à énumérer toutes ces variables, mais seulement à les traiter implicitement via un sous-problème. Parcourir explicitement toutes les colonnes pour regarder leur

coût réduit correspond alors à modifier le sous-problème leur correspondant de manière à ce que celui-ci donne accès aux colonnes de coût réduit favorable. Le branchement peut alors combiner deux aspects :

- ajout d’une contrainte dans le problème maître, ce qui implique la modification du sous-problème en modifiant l’expression du coût réduit,
- ajout d’une (ou plusieurs) contrainte(s) dans le sous-problème.

Cette dualité correspond à la décomposition d’une unique règle de branchement dans le modèle d’origine en deux parties pour le modèle décomposé. La décomposition peut lui faire correspondre une projection sur l’un ou l’autre des problèmes, voire même les deux. Nous dirons que la stratégie de branchement est *compatible* avec le sous-problème si sa projection ne détruit pas la structure du sous-problème. Ceci sera le cas si, ni la modification du coût réduit correspondant à la projection dans le problème maître, ni l’ajout d’une nouvelle contrainte correspondant à la projection dans le sous-problème, ne modifient la structure du sous-problème.

### Difficulté de la compatibilité

Modifier le sous-problème afin qu’il conserve la propriété de ne donner accès qu’aux colonnes compatibles avec les branchements et ayant un coût réduit favorable est en général le problème le plus difficile à résoudre lors de l’utilisation de méthodes de génération de colonnes. C’est souvent ce problème qui va déterminer la modélisation à utiliser pour le générateur de colonnes en éliminant les modélisations qui ne possèdent pas de schéma de branchement compatible.

Cette complexité s’ajoute alors à celle consistant à trouver un algorithme donnant un accès implicite et efficace aux colonnes non présentes dans le problème restreint. Parfois, la difficulté à mettre en œuvre un tel schéma de branchement amène à n’utiliser qu’une procédure de génération de colonnes simple sans branchement. Dans la littérature, les domaines d’application de la génération de colonnes sont encore limités et peuvent être groupés en fonction du type de sous-problème et donc souvent du type de modèle et d’algorithme utilisé pour résoudre ce sous-problème, ainsi que du type de branchement utilisé. Dans cette thèse, nous nous intéressons aux méthodes de génération de colonnes pour lesquels le sous-problème accepte un modèle de problème de plus court chemin. Pour ces problèmes, la littérature présente quasi exclusivement des générateurs à base de programmation dynamique et un branchement utilisant des règles de type *Ryan-Foster*.

L’objectif est d’obtenir non seulement la compatibilité théorique du schéma de branchement, mais également son applicabilité pratique. En effet, même si un schéma est théoriquement applicable, il sera inutilisable si chaque branchement modifie le sous-problème d’une manière telle que son temps de résolution

augmente significativement.

### Exemple de Ryan-Foster

Dans [RF81], un schéma de branchement n'ajoutant pas de contrainte au problème maître (mais seulement le sous-problème), et ne modifiant donc pas la forme du coût réduit est proposé. La proposition centrale, et qui n'a initialement pas été énoncée dans le cadre de la génération de colonnes peut être résumée de la manière suivante :

**Proposition 1** *Si  $A$  est une matrice 0 – 1, et la solution de base  $A\lambda = 1$  est fractionnelle, i.e. si au moins une des composantes de  $\lambda$  est fractionnelle, alors il existe deux lignes  $r$  et  $s$  du problème maître telles que :*

$$0 < \sum_{k:y_{rk}=1,y_{sk}=1} \lambda_k < 1$$

avec  $y_{rk}$  le coefficient de la variable  $\lambda_k$  dans le ligne  $r$ .

Une preuve de cette proposition, ainsi qu'une discussion plus générale de son application à la génération de colonnes est donnée dans [BJN<sup>+</sup>98].

Son application consiste à trouver ces deux lignes  $r$  et  $s$  du problème maître restreint et à brancher en modifiant le sous-problème correspondant à chacune des deux branches de manière telle que :

- dans une branche, seules sont utilisées des colonnes telles que  $y_{rk} = y_{sk} = 0$  ou  $y_{rk} = y_{sk} = 1$ . La somme précédemment définie vaut 1 dans cette branche.
- dans l'autre branche, seules sont utilisées des colonnes telles que  $y_{rk} = y_{sk} = 0$  ou  $y_{rk} = 0, y_{sk} = 1$  ou  $y_{rk} = 1, y_{sk} = 0$ . La somme précédemment définie vaut 0 dans cette branche.

### Cas du plus court chemin

Dans le cas où les colonnes sont le résultat d'un sous-problème de plus court chemin, il est difficile d'intégrer ces nouvelles contraintes au problème. Un schéma de branchement légèrement différent est alors utilisé consistant à trouver deux colonnes correspondant à ces lignes  $r$  et  $s$  mais ayant en plus la propriété que dans l'une des colonnes  $r$  et  $s$  sont successives dans le chemin représenté par la colonne dans le sous-problème. Le branchement correspondant consiste alors à dire que :

- dans une branche, les deux nœuds du graphe sont absents ou directement successifs,
- dans l'autre branche, les deux nœuds du graphe sont absents ou bien non successifs.

### Modification du sous-problème

Dans le cas où le sous-problème est traité par un algorithme de plus court chemin, ces deux branches correspondront à :

- dans la première branche, tous les arcs de  $r$  à d'autres nœuds que  $s$  sont supprimés, et de la même manière tous les arcs d'autres nœuds que  $r$  à  $s$  sont supprimés. Si le chemin arrive à  $r$ , il ne peut continuer que via le nœud  $s$ ,
- dans la deuxième branche, l'arc du nœud  $r$  au nœud  $s$  est supprimé.

Chacun des sous-problèmes de chacune des branches est alors de même nature que le problème d'origine, et peut être résolu en utilisant un algorithme de même complexité. De plus, il est probable que, le graphe sur lequel le chemin est cherché étant réduit, il soit en pratique plus facile à résoudre.

### Exemple de règle se projetant dans le problème maître

Les exemples de règles se projetant complètement dans le problème maître sont nombreux car ils correspondent à tous les exemples de branchement habituellement utilisés en PLNE. Par exemple, dans un problème de tournées de véhicules, il est possible de brancher sur le nombre total de véhicules utilisés si celui-ci est fractionnel dans la solution relâchée. Alors, en prenant les notations de la section 1.3.2 et  $nv$  le nombre de véhicules utilisés dans la solution relâchée (i.e. la somme totale des  $\lambda^p$ ), le branchement sera du type :

$$\sum_{p \in \Omega} \lambda^p \leq \lfloor nv \rfloor \text{ ou } \sum_{p \in \Omega} \lambda^p \geq \lceil nv \rceil$$

Il est clair que cette nouvelle contrainte devra être prise en compte dans le calcul du coût réduit des sous-problèmes. Ici, cette modification est très simple et se résume à ajouter l'opposé de la valeur duale de la contrainte de branchement.

### 2.3.3 Arbre de recherche et heuristiques

Comme toute recherche de type *Branch-and-Bound*, un arbre de recherche représentant la manière dont les décisions de branchement sont prises peut être défini. Cet arbre ne dépend pas uniquement du problème. Suivant les critères utilisés pour effectuer les branchements (e.g. choisir  $r$  et  $s$  si une règle de type Ryan-Foster est utilisée), cet arbre peut prendre des formes différentes. A tout moment, on peut classer les nœuds de cet arbre dans trois catégories :

- les nœuds déjà traités et ayant donné lieu à un branchement. Ces nœuds sont donc racine d'un sous-arbre leur correspondant,
- les nœuds déjà traités et ayant été éliminés, en fonction de divers critères,

- les nœuds ouverts.

Comme dans la PLNE sans génération de colonnes, l'ordre de création et de traitement des nœuds ouverts peut avoir une influence importante sur l'efficacité de l'algorithme. Deux critères différents influencent la définition de l'arbre de recherche ainsi que la façon dont il est parcouru :

- le choix des règles de branchement devant être utilisées à un nœud *fractionnel*,
- le choix du nœud ouvert suivant à traiter.

Nous nous référerons à ces deux choix comme *heuristique* et *stratégie de recherche*.

La littérature relative à la génération de colonnes, tout comme celle relative à la PLNE, ne contient que très peu de recherches relatives aux heuristiques et stratégies de recherche. Nous pouvons attribuer cette faible quantité de publications à la quasi absence de systèmes permettant de les étudier simplement. La dernière version de ILOG CPLEX ([ILO02a]) permet d'effectuer de telles recherches et il est à supposer que ce type de travail, beaucoup plus fréquent dans d'autres domaines comme la programmation par contraintes, devrait vite s'étendre en programmation linéaire.

Nous proposons dans cette thèse un formalisme permettant de décrire de tels types de recherches. Nous présentons également divers résultats basés sur une implantation de ce formalisme.

## 2.4 Sous-problème de plus court chemin

Dans cette thèse nous ne nous intéressons pas à tous les modèles de génération de colonnes mais uniquement à certains parmi ceux dont le sous-problème peut se formuler comme un problème de plus court chemin. Ceux-ci ne correspondent bien sûr pas l'ensemble des problèmes mais la littérature sur la génération de colonnes en comporte une forte proportion. Nous avons déjà donné plusieurs exemples d'application et nous reviendrons sur des problèmes concrets dans la partie présentant les applications.

Parmi toutes les variantes de problèmes de plus court chemin, un problème se retrouve dans de nombreux sous-problèmes de génération de colonnes. Il s'agit du problème connu sous le nom de plus court chemin avec contraintes de ressources et fenêtres de temps (SPRCTW suivant l'acronyme anglophone).

### 2.4.1 Le problème du *SPRCTW*

C'est dans [DD86] que fut introduit le problème de plus court chemin avec contraintes de ressources et fenêtres de temps comme sous problème pour la

génération de colonnes. Dans [Des88], un algorithme primal-dual pour le résoudre est présenté.

Commençons par donner une formulation complète de ce problème. Dans cette description formelle du problème, nous reprenons certaines notations de [Des88].

Soit un réseau  $G = (X, A)$ ,  $X$  représentant les nœuds de ce réseau et  $A$  les arcs. Parmi  $X$ , deux nœuds particuliers  $o$  et  $d$  représentent l'origine et la destination du chemin à trouver. Un chemin dans le réseau  $G$  est défini comme une séquence de nœuds  $(i_0, i_1, \dots, i_K, i_{K+1})$  tels que chaque arc  $(i_k, i_{k+1})$  appartienne à  $A$ . Notons que tous les chemins commencent en  $o$  (i.e.  $i_0 = o$ ) et finissent en  $d$  (i.e.  $i_{K+1} = d$ ). Un coût  $c_{ij}$  est associé à chaque arc  $(i, j)$ . Le coût total du chemin est défini comme la somme des coûts des arcs composant le chemin. Dans le problème standard, le chemin n'est pas obligatoirement élémentaire i.e. un nœud peut être présent plusieurs fois dans le chemin ( $i_j$  peut être égal à  $i_k$ , avec  $j \neq k$ ).

Tous les chemins de  $o$  à  $d$  ne sont pas tous valides.  $L$  contraintes de ressources limitent la quantité de ressources nécessaires pour atteindre chaque nœud du chemin. Pour chacune des  $L$  ressources  $l$ , chaque nœud  $i \in X$  possède une *fenêtre de validité*  $[a_i^l, b_i^l]$ . En prenant une analogie où la ressource représente le temps, il s'agit d'une fenêtre de temps où le nœud est utilisable. Pour chacune des ressources, chaque arc  $(i, j) \in A$  possède une consommation de ressource  $d_{ij}^l$ . Dans notre analogie, il s'agit du temps nécessaire pour parcourir le trajet du nœud  $i$  au nœud  $j$ . Les consommations de ressource s'accumulent le long du chemin. Avec  $T_i^l$  la quantité de ressource  $l$  nécessaire pour atteindre le nœud  $i$  en suivant le chemin de  $o$  à  $i$ , les chemins utilisant moins que  $a_i^l$  de la ressource  $l$  (i.e.  $T_i^l < a_i^l$ ) ne sont pas valides. Cependant, ils peuvent le devenir en dépensant arbitrairement la quantité manquante de ressource. Dans le cas de l'analogie avec le temps, cela correspond tout simplement à attendre que le nœud de destination soit *ouvert*. Cependant, si le chemin consomme plus que  $b_i^l$  pour atteindre le nœud  $i$ , il est irrémédiablement infaisable.

Nous pouvons considérer que  $T_o^l = 0$  pour toutes les ressources  $l \in \{1, \dots, L\}$ , et ainsi calculer  $T_i^l$  pour tout chemin faisable en utilisant  $T_{i_k}^l + d_{i_k, i_{k+1}}^l \leq T_{i_{k+1}}^l$  et  $a_{i_k}^l \leq T_{i_k}^l \leq b_{i_k}^l$  avec  $k \in \{0, \dots, K\}$  et  $l \in \{1, \dots, L\}$ . La première équation restreint l'accumulation de la ressource et la deuxième contraint les accumulations à respecter les *fenêtres de temps*.

La table 2.3 résume l'ensemble de la définition du problème de plus court chemin avec contraintes de ressources et fenêtres de temps.

### 2.4.2 Les algorithmes à base d'étiquettes

Le problème décrit dans la section précédente est communément utilisé comme sous-problème de divers modèles de génération de colonnes parmi lesquels ceux appliqués aux problèmes de tournées de véhicules et de *crew scheduling*.

<p>Trouver un chemin de coût minimal de l'origine <math>o</math> à la destination <math>d</math> (i.e. une séquence <math>(i_0, i_1, \dots, i_K, i_{K+1})</math>) et une consommation de ressource associée <math>T_{i_k}^l</math> pour chaque ressource telles que :</p>
<p>1) <math>(i_k, i_{k+1}) \in A</math> pour <math>0 \leq k \leq K</math> et <math>i_0 = o</math> et <math>i_{K+1} = d</math></p>
<p>2) <math>T_{i_k}^l + d_{i_k, i_{k+1}}^l \leq T_{i_{k+1}}^l</math> pour <math>0 \leq k \leq K</math> et <math>1 \leq l \leq L</math></p>
<p>3) <math>a_{i_k}^l \leq T_{i_k}^l \leq b_{i_k}^l</math> pour <math>1 \leq k \leq K + 1</math> et <math>1 \leq l \leq L</math></p>
<p>4) le coût total <math>(\sum_{k=0}^K c_{i_k, i_{k+1}})</math> est minimum.</p>

TAB. 2.3 – Définition du SPRCTW

De même, les algorithmes utilisés sont souvent très semblables et reposent sur les premiers modèles définis dans [DD86] et [Des88]. Tous ces algorithmes sont des algorithmes de programmation dynamique. Ils utilisent tous des chemins partiels décrits par des étiquettes. Nous décrivons ici les idées centrales à l'ensemble de ces algorithmes.

A un chemin partiel  $c$  du dépôt à une visite  $p$  est associée une étiquette correspondant au coût et quantités de ressources accumulées  $E_c^p = (C_c, D_c^1, \dots, D_c^L)$ . L'algorithme consiste à trouver de nouveaux meilleurs chemins partiels prolongeant les chemins partiels existants. Quand ceci n'est plus possible, l'étiquette au nœud final de meilleur coût correspond au meilleur chemin.

Afin de ne pas considérer tous les chemins partiels, un nouveau chemin partiel n'est conservé que s'il n'est *dominé* par aucun autre chemin partiel déjà existant. On dit qu'un chemin partiel  $c_1$  arrivant en  $p$  domine un autre chemin partiel  $c_2$  arrivant également en  $p$  s'il n'est pas possible d'obtenir un chemin à partir de  $c_2$  qui soit meilleur que n'importe quel chemin obtenu à partir de  $c_1$ . Dans le cas où les cycles sont autorisés, une condition suffisante pour que le chemin partiel  $c_1$  domine le chemin partiel  $c_2$ , tous deux chemins partiels de  $o$  au même nœud  $p$ , est :

$$C_1 \leq C_2 \quad (2.1)$$

$$D_1^l \leq D_2^l, \quad \forall l \in 1, \dots, L \quad (2.2)$$

En effet, pour toute prolongation de  $c_2$  en un chemin valide  $c_2^*$ ,  $c_1$  peut être prolongé de la manière identique pour obtenir un chemin  $c_1^*$  valide et de meilleur coût réduit que  $c_2^*$ . Le chemin partiel  $c_2$  peut donc être éliminé. Pour une démonstration et une discussion plus complète, voir [DD86].

Dans [Des88], un algorithme d'étiquetage complet est présenté, ainsi qu'une discussion sur les différentes implantations possibles.

### 2.4.3 Autres types de contraintes

Le modèle et l'algorithme de SPRCTW de la section précédente permettent de modéliser et résoudre de nombreux sous-problèmes apparaissant en génération de colonnes. Certaines contraintes définissant la validité des colonnes, même si elles ne correspondent pas directement à une ressource concrète, peuvent être intégrées au schéma de résolution sous une forme indirecte.

Par exemple, l'existence de plusieurs origines différentes possibles pour les chemins peut être intégrée au modèle en utilisant plusieurs SPRCTW différents, chacun correspondant à une origine différente. C'est ce qui est généralement proposé pour le problème de tournées de véhicules avec plusieurs dépôts (voir la section 5.3).

Cependant, les applications réelles mettent en œuvre des contraintes résultant de réglementations qui parfois se révèlent trop complexes pour être intégrées à l'algorithme à base d'étiquettes. Pendant longtemps, la seule méthode alors disponible consistait à générer des colonnes et vérifier leur validité (voir [FPW96, FPW97, FP98]). Quand le nombre de colonnes était prohibitif, divers critères heuristiques limitaient l'ensemble de celles devant être testées. Aujourd'hui, la programmation par contraintes se présente comme une alternative prometteuse. Celle-ci a l'avantage d'être théoriquement ouverte à tous ces nouveaux types de contraintes.

### 2.4.4 Algorithmes de programmation par contraintes

Déjà quelques articles présentent comment des méthodes de programmation par contraintes sont utilisées pour résoudre des problèmes de plus court chemin contraints. Nous donnons ici quelques idées sur les modèles utilisés et la manière dont la programmation par contraintes permet de résoudre plus efficacement ces problèmes.

#### Modèles

A chaque nœud  $i$  est associée une variable contrainte `next[i]`, dont le domaine est l'ensemble des indices des nœuds qui peuvent être successeurs directs du nœud  $i$ . A chaque ressource est également associé un tableau de variables `cumul`, telles que pour un nœud  $i$ , la variable `cumul[i]` représente l'accumulation de ressource jusqu'au nœud  $i$ . Les diverses contraintes sont ensuite formulées en utilisant ces variables.

Pour chaque ressource, une contrainte sur l'accumulation de la ressource est utilisée. Cette contrainte assure que si

`next[i] = j`

alors

```
cumul[j] >= cumul[i] + distance(i,j)
```

Nous nous référerons à cette contrainte en utilisant le nom `pathlength`, en suivant la nomenclature utilisée dans l'environnement de programmation par contraintes ILOG Solver [ILO02c].

Les fenêtres de faisabilité sont ensuite définies par des contraintes sur les variables. Par exemple :

```
a[i] <= cumul[i] <= b[i]
```

Enfin, la programmation par contraintes permet de définir l'objectif simplement en donnant l'expression des variables contraintes à optimiser. Par exemple :

```
cumul1[endIndex] + 2*cumul2[endIndex]
```

Un modèle contenant une combinaison de tous ces éléments permet de définir aisément un problème de plus court chemin avec contraintes de ressources et fenêtres de temps correspondant au modèle de la section 2.4.1. Ensuite, d'autres contraintes très variées peuvent être ajoutées, comme par exemple :

```
ifthen( next[i] == j, next[j] != k )
```

Aussi, les environnement de programmation par contraintes ouverts comme ILOG Solver [ILO02c] permettent de définir simplement de nouveaux types de contraintes. Si une réglementation ne peut pas être exprimée comme combinaison des contraintes pré-définies, une contrainte spécifique peut être utilisée.

## Résolution

Afin de résoudre les problèmes de plus court chemin, la programmation par contraintes offre plusieurs possibilités. Celles-ci correspondent aux différents niveaux de filtrage pouvant être utilisés. La littérature présente deux niveaux de filtrage. Nous proposerons dans un chapitre ultérieur d'utiliser un troisième niveau.

Tout d'abord, un premier niveau simple consiste à propager de manière très simple les bornes des ressources accumulées. Quand une variable `next[i]` s'instancie à une valeur `j` (i.e. quand `j` reste la seule valeur possible dans le domaine de `next[i]`), alors la borne inférieure de `cumul[j]` est modifiée pour être égale à la borne inférieure de `cumul[i]` plus la distance `distance(i,j)`. Ainsi, quand toutes les variables sontinstanciées, l'affectation obtenue respecte les contraintes d'accumulation de ressources. Si certaines fenêtres de temps, ou autres contraintes spécifiques, ne sont pas vérifiées, un échec est obtenu auparavant. Il est facile de

voir que cette propagation de bornes ajoutée à d'autres propagations de bornes obtenues via d'autres contraintes permet une résolution plus efficace que celle consistant à générer puis tester.

Un deuxième niveau de propagation consiste à prendre en compte une ressource globalement. Pour un nœud donné  $i$ , la borne inférieure peut être obtenue en tenant compte du chemin le plus court (vis-à-vis de cette ressource uniquement) depuis l'origine  $o$  jusqu'au nœud  $i$ . Chaque contrainte `pathlength` peut donc utiliser un algorithme de plus court chemin pour établir de meilleures bornes aux accumulations de ressources. L'arbre de recherche défini par cette propagation sera encore plus réduit que celui défini par la propagation précédente. Cependant, le temps de calcul pour effectuer le filtrage peut être plus long. Une méthode similaire est proposée dans [JKK<sup>+</sup>99].

## 2.5 Méthodes d'accélération

Les méthodes de génération de colonnes complètes comme le *Branch-and-Price* ont des complexités non polynomiales. Le temps d'exécution réel peut donc être prohibitif et rendre une implantation simple inutilisable. Comme nous l'avons souligné dans l'introduction, cette raison doit expliquer pourquoi ces méthodes sont restées quasi inutilisées durant plusieurs décennies. Récemment, de nombreuses méthodes visant à réduire les temps d'exécution ont été proposées. Nous présentons plusieurs de ces propositions dans les sections suivantes, tout d'abord celles propres à la génération de colonnes, puis celles propres aux problèmes de plus court chemin, et enfin celles mettant en jeu les deux aspects.

Auparavant, il convient de revenir sur une idée reçue répandue qui consiste à penser que :

90 % du temps d'exécution étant passé dans la résolution du sous-problème, l'effort doit être consacré quasi uniquement à réduire le temps d'exécution de celui-ci.

Imaginons que la répartition des temps d'exécution entre le problème maître et le sous-problème pour un problème concret soit effectivement du type 10 % / 90 % . Notons alors qu'une modification de la manière dont fonctionne le problème maître résultant en une réduction de 10 % du nombre d'appels au sous-problème résulte dans la même réduction du temps total d'exécution qu'une réduction de 10 % du temps d'exécution du sous-problème. Il convient donc de ne pas centrer les efforts sur un seul de ces deux aspects.

### 2.5.1 Méthodes d'accélération du problème maître

Une liste de méthodes d'accélération pour les applications aux problèmes de *tournées de véhicules* et de *Crew Scheduling* est donnée dans [DDS01]. Nous reprenons ici une description des méthodes d'accélération sans les lier à un type d'application particulier.

#### Solutions initiales et variables d'écart

La génération de nouvelles colonnes nécessite l'existence d'une solution courante, et de l'information duale correspondante, pour fonctionner. Hors, le modèle utilisé dans le problème maître restreint ne permet pas toujours d'obtenir simplement une telle solution initiale évidente pour alimenter l'algorithme. En effet, si le problème est, par exemple, un problème de *packing* pur, ne contenant que des contraintes du type :

$$\sum_i \lambda_i x_{i,j} \leq d_j$$

l'algorithme peut démarrer avec une solution initiale où toutes les variables sont implicitement nulles, se qui équivaut à un problème restreint où aucune colonne n'est présente. Ceci n'est pas le cas avec des contraintes de type *covering* :

$$\sum_i \lambda_i x_{i,j} \geq d_j.$$

Dans ce dernier cas, les  $x_i$  n'étant pas disponibles au début de la recherche, il n'existe pas de solution évidente. Comme pour la programmation linéaire, deux possibilités existent :

- utiliser une solution initiale obtenue avec une autre technique d'optimisation,
- utiliser des variables d'écart.

**Solution Initiale** Pour une grande majorité des problèmes traités avec les techniques de génération de colonnes, une solution de qualité moyenne peut être obtenue en utilisant un algorithme spécifique simple. Pour le problème de routage de véhicules, un algorithme glouton créant successivement des tournées en leur ajoutant des visites possibles suivant un critère simple peut être utilisé. Quand ceci n'est plus possible, une nouvelle tournée est utilisée. La solution ainsi obtenue, valide, si le nombre maximal de tournées ou de véhicules n'est pas limité, peut alors être utilisée comme solution initiale.

**Variation d'écart** Lorsqu'il n'est pas possible d'utiliser une méthode simple afin d'obtenir une solution initiale, la méthode des variables d'écart peut être

utilisée de la même façon que pour la PLNE. Pour chaque contrainte  $j$  du type :

$$\sum_i \lambda_i x_i \geq d_j.$$

Une variable d'écart positive  $s_j$  est ajoutée ne participant qu'à la contrainte  $j$ , de borne supérieure égale à  $d_j$  et de fort coût dans l'objectif. Le problème complet devient :

$$\begin{aligned} \min \quad & \sum_i c_i x_i + \sum_j c'_j s_j, \\ & \sum_i \lambda_i x_i + s_j \geq d_j, \forall j \end{aligned}$$

Le choix du coût  $c'_j$  est important car il doit être suffisamment important pour que cette variable ne fasse pas partie de la solution finale. Dans le cas le plus fréquent où  $d_j = 1$ , il peut être choisi en fonction d'une borne supérieure sur les coûts des variables intervenant dans cette contrainte  $j$ . Par exemple, dans le cas du problème de tournées de véhicules, les contraintes de couverture des visites contraignent la somme des variables représentant les tournées passant par une visite  $j$  à être supérieure à 1. On peut alors donner à la variable d'écart un coût égal à un majorant des coûts des tournées.

**Solutions intermédiaires** Une méthode intermédiaire aux deux méthodes précédentes consiste à utiliser une solution initiale pour une partie du modèle maître et des variables d'écart pour la partie du problème où cette solution n'est pas valide.

Par exemple, la solution au problème de routage de véhicules obtenue avec l'algorithme simple présenté auparavant, et qui n'est pas valide si le nombre de véhicules est limité à une valeur inférieure à celle utilisée dans cette solution, peut être utilisée si une variable d'écart négative est utilisée dans la contrainte limitant le nombre de véhicules. La valeur du coût de cette variable doit être choisie correctement.

### Qualité de la solution duale initiale

Le choix de la solution initiale n'est pas indifférent. Même si la solution finale est toujours de même valeur, différentes solutions initiales permettent d'aboutir à l'optimum relâché en un nombre d'itérations plus ou moins important.

Illustrons ce fait sur la situation similaire de la méthode de *Branch-and-Cut*. Il est évident que si la solution initiale utilisée est valide pour l'ensemble du

problème, aucune coupe ne sera générée et l'algorithme aboutira en une seule étape. Ce cas est bien sûr exceptionnel. Plus généralement, le choix de la solution initiale, ainsi que les critères de sélection des coupes devant être ajoutées sont deux paramètres influençant le *chemin* suivi par les solutions correspondant aux itérations successives à l'intérieur du polytope complet. Chacun de ces deux paramètres agit sur ce chemin et sa longueur, i.e. le nombre d'itérations de l'algorithme.

La même situation existe dans le cas de la génération de colonnes. Le choix de la solution initiale ainsi que la méthode de sélection des variables devant être ajoutées au problème restreint sont deux paramètres ayant un impact sur la série de solutions obtenues. Il n'existe cependant pas de technique systématique pour choisir la meilleure solution duale initiale (celle permettant d'obtenir l'optimum relâché en un nombre minimal d'itérations). L'idéal serait évidemment de démarrer avec la solution finale ... Cependant, nous pouvons supposer qu'il est préférable de choisir une solution duale proche de la solution duale finale estimée. C'est ce que nous essayons de faire quand nous utilisons un algorithme pour fournir une solution primale la plus satisfaisante possible (cas de l'exemple du problème de tournées). Le choix du coût des variables d'écart permet, d'autre part, d'influer sur la solution duale initiale.

### Stabilisation des variables d'écart

Il a été montré qu'en plus d'un choix adéquat pour les bornes et coûts des variables d'écarts, une procédure les actualisant peut permettre d'obtenir plus rapidement la solution optimale relâchée. Cette méthode est connue sous le nom de *stabilisation* des variables d'écart. Plusieurs variantes sont possibles définissant quand et comment sont actualisés les bornes et coûts des variables d'écart. Une description assez complète de l'utilisation de la stabilisation dans le cadre de la génération de colonnes est donnée dans [dMVDH99] et [EY99].

### Ajout de colonnes par paquets

Dans la méthode du simplexe, les variables entrent en base une par une. Dans le cas de la génération de colonnes, ajouter des variables au problème restreint consiste à les ajouter à la liste de variables candidates pour entrer en base. Aucune raison propre au fonctionnement de la programmation linéaire n'incite particulièrement à ajouter plusieurs colonnes à cette liste à chaque itération. Cependant, l'algorithme utilisé dans le sous-problème est en général coûteux en terme de temps, de manière que si il permet d'obtenir plusieurs colonnes de coût-réduit favorable pour une seule exécution, il peut être avantageux d'utiliser ces différentes colonnes et de les ajouter *par paquets* au problème maître restreint. Ajouter plusieurs colonnes de coût réduit favorable au problème maître restreint

ne signifie en aucun cas que toutes ces colonnes entreront en base. En effet, ce qui est certain est qu'une des colonnes ajoutées entrera en base, ce qui peut avoir pour effet de modifier la solution duale courante. Le coût réduit d'une ou plusieurs des autres colonnes fournies peut alors ne plus être favorable. Cependant plusieurs situations sont possibles :

- si les colonnes du paquet fourni au problème maître restreint sont suffisamment variées, il est possible que plusieurs d'entre elles finissent par entrer en base de manière simultanée. Ceci est en particulier probable au début de la phase de génération.
- d'autre part, il est possible que la solution duale obtenue après l'ajout de la meilleure colonne soit toujours favorable à l'ajout d'une des autres colonnes qui viendrait remplacer la première. Même si cette colonne avait été obtenue par une autre exécution de l'algorithme de génération, l'ajout de plusieurs colonnes simultanément permet de réduire le nombre d'exécutions de ce sous-problème.

### Elimination de colonnes

Même si le nombre de colonnes ajoutées au problème maître restreint est très inférieur au nombre total de colonnes dans le problème complet, il est possible qu'il atteigne une valeur importante et que ceci ralentisse la résolution des problèmes linéaires de manière significative. Parmi les colonnes présentes dans le problème restreint, seul un très petit nombre d'entre elles sont utilisées dans la solution relâchée courante. Parmi toutes les autres colonnes, certaines seront utilisées dans la solution entière ou dans d'autres solutions relâchées ultérieurement. Leur entrée en base résultera d'une règle de branchement ou d'une coupe. Enfin, il y a de nombreuses colonnes qui ne seront jamais plus utilisées.

La même situation existe dans le cas du *branch-and-cut*. Certaines coupes ajoutées à un nœud alors qu'elles étaient violées ne sont plus utilisées dans la solution duale. Il convient donc de savoir retirer certaines de ces coupes du problème restreint. Un conseil souvent donné est d'être conservateur lorsqu'il s'agit d'ajouter des coupes et libéral quand il s'agit de les enlever, ceci afin de conserver un problème restreint de petite taille facilement manipulable. Ceci est en particulier vrai quand une recherche arborescente est menée et qu'il faut pouvoir passer d'un nœud à un autre de manière efficace.

Il convient donc dans le cas de la génération de colonnes, de pouvoir garder une taille de problème restreint raisonnable en éliminant certaines des colonnes. Il est cependant difficile en général de savoir quelles sont parmi les colonnes présentes celles qui ne seront jamais utilisées (sauf dans le cas où ceci peut être démontré, e.g. en utilisant le coût réduit comme dans la section 2.5.1). En général, les méthodes de sélection des colonnes à retirer du problème restreint se limitent

à en choisir certaines parmi celles de coût réduit non favorable. C'est ce qui est proposé par exemple dans [JMV99a].

### Génération approximative et complète

La solution du problème relâché ne peut être utilisée comme borne duale que si aucune des variables non présentes dans le problème maître relâché n'a un coût réduit favorable. Une itération principale de l'algorithme de génération de colonnes consiste donc à chercher si une telle colonne existe et dans l'affirmative, à l'ajouter au problème relâché, jusqu'à ce que ceci ne soit plus possible. Dans le cas où une ou plusieurs colonnes de coût réduit favorable existe, l'algorithme n'impose pas que ce soit celle de coût réduit le plus favorable qui soit ajoutée. La seule condition nécessaire imposée à l'algorithme de génération est de pouvoir assurer la non existence d'une telle colonne quand ceci est le cas.

Pour un problème donné, il existe très souvent un algorithme approximatif permettant d'exhiber une colonne de coût réduit favorable (i.e. respectant, en plus des contraintes du sous-problème proprement dites, une contrainte sur le coût réduit), de complexité inférieure aux algorithmes, dit *complets*, permettant d'exhiber la colonne de coût réduit optimal. Une méthode efficace consiste alors à utiliser l'algorithme de complexité moindre tant que celui-ci permet d'obtenir des colonnes de coût réduit favorable et à utiliser ensuite l'algorithme complet uniquement afin d'assurer la non existence d'autres colonnes. Si l'algorithme complet trouve une colonne favorable nouvelle, l'algorithme heuristique est alors utilisé de nouveau. La figure 2.2 montre une itération de génération complète.

Dans la littérature, différentes procédures de ce type sont proposées. Elles restent cependant assez simple :

- ne mettant en œuvre en général que deux types de générateurs différents,
- étant identiques à tous les niveaux de l'arbre de recherche.

Dans cette thèse nous montrons comment des schémas plus complets peuvent être facilement et efficacement utilisés.

### Colonnes pré-générées

Une méthode parfois utilisée pour obtenir rapidement des colonnes favorables consiste à générer au préalable une quantité importante de colonnes valides et considérées comme particulièrement intéressantes. Ensuite, à chaque nœud de l'arbre de recherche, avant de faire appel à un générateur complet proprement dit, une itération sur ces colonnes permet de

- vérifier quelles sont celles qui sont compatibles avec l'ensemble des décisions de branchement prises jusqu'au nœud courant,
- calculer leur coût réduit.

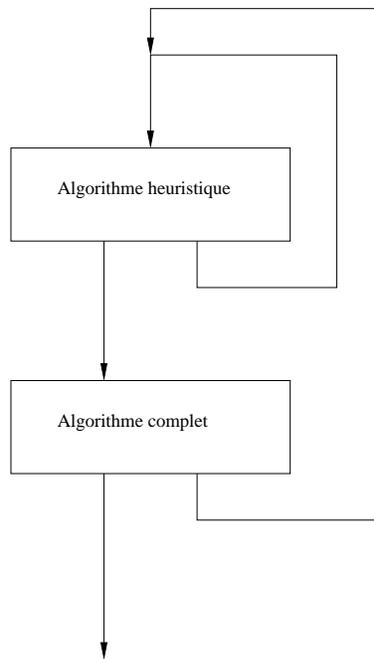


FIG. 2.2 – Illustration de la possible interaction entre génération heuristique et génération complète

Cette méthode permet d'offrir un compromis entre un algorithme de type PLNE où toutes les colonnes seraient présentes dans le problème et un algorithme de type *Branch-and-Price* où un générateur est fréquemment interrogé. Une quantité importante de temps de calcul est dépensée au début à générer ce paquet de colonnes favorables, mais ce temps peut être économisé par la suite.

Quand aucune colonne de ce paquet pré-généré n'est favorable, l'algorithme de *Branch-and-Price* doit théoriquement faire appel à un algorithme de génération complet. Certains articles éliminent volontairement cet appel et obtiennent ainsi un algorithme de *Branch-and-Price heuristique* qui, même s'il ne permet plus d'assurer l'optimalité de la solution trouvée, ni d'obtenir des bornes inférieures, peut permettre cependant de trouver plus rapidement des solutions acceptables, le temps dédié au générateur complet étant économisé.

Ce type d'algorithme est utilisé dans [FPW96, FPW97, FP98].

### Partage de colonnes entre nœuds

La méthode précédente consistant à pré-générer un grand nombre de colonnes peut être hybridée avec les méthodes consistant à générer dynamiquement les colonnes quand cela est nécessaire. En effet, au lieu d'utiliser un grand nombre de colonnes toutes calculées au préalable, l'ensemble des colonnes générées en un nœud quelconque de l'arbre de recherche peut être utilisé comme réserve de colonnes utilisables dans les autres nœuds. Chaque nouvelle colonne générée à un nœud est ajoutée à cette réserve de colonnes. Avant de lancer l'algorithme complet à un nœud, cette réserve est interrogée de la même manière que dans la méthode précédente. Cette méthode est connue sous le nom de *pool pricing*.

### Arrêt prématuré de la génération de colonnes de coût entier

Cette méthode est à notre connaissance introduite par [VBJN94] et [Van98] pour le problème de découpe unidimensionnelle, et reprend une idée de [Far90]. Elle s'applique aux problèmes où les coûts sont contraints à ne prendre que des valeurs entières.

Dans ce cas, il n'est pas toujours nécessaire de résoudre le problème relâché jusqu'à l'optimalité (et donc de compléter intégralement la génération de colonnes de coût réduit favorable) pour obtenir une borne inférieure (dans le cas de la minimisation) du problème entier. A un nœud de la recherche, si  $z_{LP}$  représente la valeur du problème relâché,  $\bar{r}\bar{c}$  le meilleur coût réduit parmi les colonnes non encore ajoutées et  $z_{IP}$  la valeur optimale recherchée du problème entier pour le nœud, nous avons la relation suivante :

**Théorème 1** Si  $\lceil z_{LP} \rceil = \lceil z_{LP} / (1 - \bar{r}\bar{c}) \rceil$ , alors  $z_{IP} \geq \lceil z_{LP} \rceil$

Une preuve de ce résultat est donnée dans [VBJN94].

Une conséquence de ce résultat est que la résolution du nœud peut s'arrêter quand nous avons  $\lceil z_{LP} \rceil = \lceil z_{LP}/(1 - \bar{r}c) \rceil$ . De même, pour chaque autre nœud de la branche correspondante, quand la valeur du problème relâché passe au dessous de cette valeur seuil, il peut être intéressant de brancher immédiatement. En effet, le théorème assure qu'il sera finalement nécessaire de brancher.

### Réduction de bornes par coût réduit

Cette opération est largement utilisée en programmation linéaire. Elle consiste à changer la borne inférieure ou supérieure d'une variable en fonction de son statut dans la base et de la valeur de son coût réduit. Elle a été proposée dans le domaine de la génération de colonnes dans [JRT94] et ensuite dans [JT97a, JT97b].

Cette réduction peut se définir, pour un problème de minimisation, et en reprenant l'exemple donné dans [JRT94], de la manière suivante :

Si pour un nœud de la recherche, où la borne inférieure locale (borne duale) est notée  $ldb$ , et la borne supérieure globale (borne primale, i.e. meilleure solution trouvée) est notée  $gpb$ , une variable booléenne  $x_e$  n'est pas en base et si son coût réduit  $cr_e$  est tel que  $ldb + cr_e > gpb$ , alors cette variable peut être *mise à zéro* i.e. elle ne participera à aucune solution dans cette branche. Une autre formule de même type est utilisable pour *mettre à un* une variable. On parlera alors de *setting*.

Dans le cas où cette opération est effectuée au niveau du nœud racine de l'arbre de recherche, le résultat est valable pour l'ensemble de l'arbre, et on parlera alors de *fixing*.

### Réduction de bornes par implication logique

Cette opération consiste également à déduire de nouvelles bornes pour certaines variables, mais cette fois ci avec des déductions logiques utilisant les bornes d'autres variables. Ceci peut être vu comme une opération de filtrage simple.

## 2.5.2 Méthodes d'accélération du problème de plus court chemin seul.

De même que nous avons présenté dans la section précédente des techniques d'accélération propres à la génération de colonnes, nous présentons ici des méthodes propres au problème de plus court chemin. Toutes ces accélérations ne sont pas nécessairement liées à la nature de l'algorithme utilisé pour la résolution mais peuvent être utilisables indépendamment de ce dernier. Par contre, nous ne considérons pas ici les méthodes d'accélération tenant compte de l'aspect sous-problème. Ces accélérations sont présentées dans la prochaine section.

### Raisonnement sur les fenêtres de temps

Des raisonnements simples sur le graphe utilisé pour générer de nouveaux chemins permet de réaliser certains pré-traitements réduisant le problème à traiter. Ces raisonnements, simples à mettre en œuvre, sont utilisés dans la majorité des implantations et étaient déjà proposés dans [DD86].

Un exemple de ces éliminations est :

Si  $a_i^l + d_{i,j}^l > b_j^l$  alors l'arc  $(i, j)$  ne pourra jamais faire partie d'une quelconque solution vu qu'il ne peut faire partie d'aucun chemin valide. En éliminant cet arc au préalable, on peut éviter à un algorithme de type programmation dynamique d'avoir à utiliser des étiquettes inutiles.

### Limitation du nombre de chemins complets

Le temps d'exécution de la résolution du problème de plus court chemin peut être limité en arrêtant l'algorithme quand un nombre minimal, initialement fixé, de chemins valides et de coût réduit favorable ont été trouvés. Cette méthode a pour avantage principal d'assurer que si au moins un chemin valide de coût réduit favorable existe, alors au moins un chemin valide de coût réduit favorable sera obtenu. Au contraire, l'optimalité n'est plus garantie. Nous avons cependant vu dans la section 2.5.1 que ceci peut ne pas être un problème en génération de colonnes. Un inconvénient véritable est la difficulté de contrôler à priori le temps d'exécution avec ce type de réduction.

Cette méthode est utilisée dans [Lar99] et [Erd99]. Nous l'utilisons également dans le chapitre 5.

### Réduction heuristique

L'optimalité de la solution n'étant pas nécessaire à tout moment, plusieurs autres réductions heuristiques du sous-problème de plus court chemin ont été proposées. La version heuristique de l'algorithme peut être utilisée tant que des chemins valides et de coût réduit favorable sont trouvés si la version complète est utilisée en fin de traitement du nœud.

**Limitation du nombre d'étiquettes dominantes à un nœud** Une manière simple de réduire heuristiquement le temps d'exécution consiste à limiter le nombre d'étiquettes conservées à un nœud. Des étiquettes non-dominées sont alors éliminées. Cette limitation ne garantit alors ni l'optimalité de la meilleure solution trouvée, ni l'absence de solution si aucune n'est trouvée. Cependant, elle permet d'éviter l'explosion du nombre d'étiquettes.

**Discrétisation des dimensions** Une technique qui permet d'obtenir indirectement un résultat similaire à la technique précédente consiste à discrétiser les consommations de ressources possibles. Par exemple, dans le cas du problème de VRP, arrondir toutes les distances entre les noeuds. Ainsi, de deux étiquettes pour lesquelles l'accumulation arrondie est identique peut n'être gardée qu'une seule. Par contre, cette limitation du nombre d'étiquettes ne permet pas de définir précisément la limitation à chaque noeud et peut nécessiter une paramétrisation plus fine.

### 2.5.3 Méthodes d'accélération mettant en jeu les deux problèmes

Dans cette section sont présentées des méthodes d'accélération mettant en jeu à la fois la génération de colonnes et le sous-problème de plus court chemin.

#### Réduction du graphe

De même qu'il a été possible de déduire que certains noeuds ou arcs du graphe ne seraient pas utilisés dans aucune solution du problème de plus court chemin en utilisant uniquement des déductions propres à la définition du problème de plus court chemin, l'information duale venant du problème maître peut être utilisée afin d'éliminer d'autres éléments du graphe. Cette méthode a été introduite dans [RGP02] pour un environnement de programmation par contraintes et également utilisée dans [CDP02] dans le cadre de la génération de colonnes.

Illustrons cette technique en utilisant l'exemple du problème de tournées de véhicules tels que présenté dans la section 1.3.2. Rappelons que nous avons utilisé comme notations :

- $\pi_i$  représente la valeur duale associée à la couverture de la visite  $i$ ,
- le coût réduit devant être minimisé est  $\sum_{(i,j) \in p} (c_{ij} - \pi_j) - \pi_0$
- $cr_{ij} = c_{ij} - \pi_j$  représente la *part* du coût réduit du chemin correspondant à l'arc  $(i, j)$ ,
- $succ(i)$  l'ensemble des successeurs possibles de  $i$ .

**Proposition 2** Soient deux noeuds  $i$  et  $j$  du graphe, tel que  $(i, j) \in A$ , si :

$$\forall k \in succ(j), cr_{ij} + cr_{jk} > cr_{ik}$$

alors l'arc  $(i, j)$  peut être éliminé de  $A$ .

Cette configuration est représentée dans la figure 2.3.

En effet, il est facile de voir que s'il existait une solution optimale passant par le chemin partiel  $(i - j - k)$ , alors un chemin de meilleur coût pourrait être

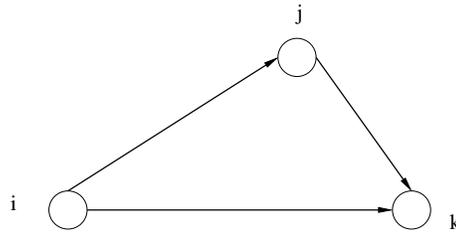


FIG. 2.3 – Réduction du graphe utilisant le coût réduit

obtenu en remplaçant ce chemin partiel par le chemin partiel  $(i-k)$ , et l'hypothèse précédente utilisée pour montrer que le chemin serait de meilleur coût réduit. Bien que cette méthode utilise le coût réduit pour réduire le graphe, elle est de nature différente de celle présentée dans 2.5.1.

### Itérativité du sous-problème

Au delà de toutes les méthodes d'accélération précédemment présentées, et consistant à simplifier une instance de sous-problème ou à recourir moins souvent à la génération de nouvelles colonnes, l'aspect itératif du sous-problème peut être exploité. Deux niveaux d'itérativité co-existent.

Tout d'abord, pour un même nœud, le sous-problème est résolu plusieurs fois, chaque recherche ne diffère de la précédente que par le schéma de coût, et en particulier par les valeurs duales provenant du problème maître.

D'autre part, la première exécution à un nœud ne diffère de la dernière exécution du nœud père que par ce schéma de coût et par quelques contraintes résultant de la règle de branchement utilisée.

Il peut alors être intéressant de réutiliser les résultats de l'exécution précédente et/ou l'état complet de l'algorithme à la fin de l'exécution précédente pour initialiser le nouveau problème. Plus concrètement, dans le cas où le sous-problème est résolu en utilisant une technique d'étiquetage, il est possible de réutiliser les étiquettes obtenues à la fin de l'exécution précédente. Il suffit alors de vérifier leur validité vis-à-vis d'éventuelles nouvelles règles, et d'actualiser leur coûts. Le nombre d'itérations ensuite nécessaires pour obtenir un ensemble optimal d'étiquettes est généralement réduit.

## 2.6 Coupes : *Branch-and-Price-and-Cut*

L'utilisation de *coupes* en PLNE est fréquente et souvent efficace comme nous l'avons commenté dans la section 1.1.2. Il est donc logique de vouloir utiliser des

coupes également dans le cadre de la génération de colonnes. Une coupe est simplement une contrainte linéaire du problème complet non ajoutée au programme linéaire initial. Il existe deux usages principaux des coupes :

- pour améliorer la qualité de la solution relâchée,
- pour le cas où le nombre de contraintes est trop important.

Ces deux cas se présentent également en génération de colonnes. Quand des coupes sont utilisées dans une procédure de *Branch-and-Price* on parle alors de *Branch-and-Price-and-Cut*.

### 2.6.1 Coupes pour le problème relâché

Nous avons vu que la génération de colonnes au seul nœud racine ne permet pas d'assurer qu'une solution entière va être obtenue. La solution optimale relâchée peut cependant être utilisée comme borne duale pour le problème entier d'origine. Cette situation se répète de manière identique pour tout sous arbre de l'arbre de recherche. De la même manière qu'en PLNE, il est possible de *couper* cette solution relâchée. L'utilisation d'une coupe peut offrir plusieurs types d'avantages :

- en fonction de la coupe utilisée, la nouvelle solution relâchée peut faire apparaître une solution dont l'intégralité est moins violée,
- la borne duale obtenue avec la nouvelle solution relâchée ne peut être que meilleure ou égale (problème plus contraint),
- finalement, l'information pouvant être obtenue de la nouvelle solution peut être plus significative au moment de réaliser un branchement.

A notre connaissance, des coupes de ce type n'ont été utilisées en génération de colonnes sur des modèles décomposés que dans le cas de problèmes de tournées de véhicules. Dans [Koh95], l'utilisation de plans de coupes est largement présentée. Des résultats précis sur des instances reconnues sont donnés dans [CR99], ainsi qu'une parallélisation de l'algorithme de séparation. Nous présentons plus longuement ce type de coupes dans le chapitre 5. Nous présenterons dans le chapitre 7 différentes coupes utilisées de façon originale dans le cadre de la génération de colonnes.

### 2.6.2 Coupes sur la validité

L'autre utilisation classique des coupes a déjà été plus étudiée dans le cadre de la génération de colonnes. Dans [JMV99a, JMV99b], un problème d'affectation de fréquences est résolu en utilisant des méthodes de génération de colonnes. Pour ce problème, le nombre de colonnes et de contraintes sont tous deux trop importants pour que toutes soient mis dans le problème linéaire dès le début. Seules quelques contraintes sont alors prises en compte. Quand une solution entière est trouvée,

elle peut n'être cependant pas valide si elle viole une des contraintes non encore ajoutées. Ces contraintes sont alors ajoutées.

### 2.6.3 Gestion des coupes en génération de colonnes

Quand toutes les variables sont initialement présentes, les coupes peuvent simplement, mais complètement, être ajoutées et retirées du problème. Quand de nouvelles variables peuvent également être générées, la situation est plus compliquée. En effet, les variables non explicitement présentes dans le problème maître peuvent avoir implicitement des coefficients non nuls dans ces coupes.

Deux choses sont alors à prendre en considération :

- la génération de colonnes doit être modifiée pour prendre en compte la partie de coût réduit qui correspond à l'éventuel coefficient qu'aurait la colonne solution dans chacune des coupes présentes dans le problème maître,
- les coupes doivent être modifiées quand de nouvelles variables sont ajoutées de manière à prendre en compte ces nouvelles variables.

## Conclusions

Dans ce chapitre, nous avons présenté un état de l'art des techniques récemment proposées pour la mise en œuvre des méthodes de génération de colonnes. L'émergence de ces différentes méthodes accélérant la procédure traditionnelle de génération de colonnes (dont, entre autres, l'utilisation d'algorithmes spécifiques et efficaces pour la résolution des sous-problèmes) a été, selon nous, le principal facteur de leur retour au premier plan durant les dernières années.

Cependant, il doit être noté que seule une partie très faible de ces techniques ont été ré-utilisées pour des problèmes différents de ceux sur lesquels elles ont été initialement proposées. Il nous semble que la raison en est l'absence d'un formalisme fédérateur. Nous proposons dans la partie suivante un formalisme générique permettant la description globale des modèles décomposés ainsi que de leur procédure de résolution. Nous montrerons ensuite dans la dernière partie, en utilisant des exemples variés, comment ce formalisme permet à une contribution nouvelle d'être facilement applicable à des problèmes divers.

Deuxième partie

Modélisation et Recherche  
Générique



Peu de travaux utilisant des méthodes de génération de colonnes généralisent leurs résultats sur des applications autres que celle sur laquelle elles ont été proposées. Parmi les raisons à cette absence, il semble que la principale soit l'absence apparente de relations entre les différents modèles proposés pour les différentes applications.

Nous proposons ici certains concepts permettant de décrire de manière homogène des problèmes différents mais tous pris sous la forme décomposée propre à la génération de colonnes. Ce chapitre reprend en partie des travaux présentés dans le cadre de [Cha00a, Cha00b, Cha02b].

Parmi les éléments intéressants à généraliser, citons :

- les modèles utilisés pour décrire les problèmes, tant au niveau du problème maître que du sous-problème,
- la manière dont sont réalisées les interactions entre les deux problèmes, i.e. la définition automatique des colonnes correspondant à une solution d'un sous-problème, et l'actualisation des sous-problèmes en fonction d'une solution du problème maître,
- la procédure utilisée pour chercher des solutions, i.e. l'heuristique utilisée.

Cette partie comprend deux chapitres. Le premier propose un formalisme pour décrire les modèles de génération de colonnes qui essaie de répondre aux deux premiers éléments précédents. Ce formalisme est illustré par des exemples de modèles pour des problèmes simples résolus par des méthodes de génération de colonnes. Il sera également utilisé dans la partie suivante pour décrire les modèles que nous avons utilisés. Le second chapitre propose d'utiliser le formalisme des *goals* pour décrire la recherche.

Notre objectif en proposant ces deux formalismes a été double. D'une part théorique, il permet une description plus générique des problèmes, plus facile à partager entre différentes personnes, mais également pratique, car il a permis une implantation d'un système plus générique de procédures de génération de colonnes et de coupes. Cette implantation est décrite dans l'annexe A.



# Chapitre 3

## Modélisation générique

Dans ce chapitre, nous proposons un formalisme original permettant de décrire de manière générique les problèmes traitables par génération de colonnes. Ce formalisme permet également de définir de manière implicite les interactions entre le problème maître et les différents sous-problèmes.

### 3.1 Modèles

Nous reprenons les notations utilisées lors de l'introduction de problèmes d'optimisation dans la section 1.1.

#### 3.1.1 Sous-problème

Le sous-problème consiste à générer de nouvelles colonnes devant être ajoutées au problème maître restreint. Les colonnes pouvant participer au problème maître sont alors solutions d'un problème qui contient des contraintes, des variables contraintes et éventuellement un objectif.

**Définition 10** *Le sous-problème est ou bien un problème de satisfaction de contraintes ou bien un problème d'optimisation.*

**Exemple :** Dans le problème de découpe à une dimension de la section 1.3.1, le sous-problème consiste à générer de nouveaux patrons de découpe de coût réduit négatif. Des variables  $x_i$  représentent combien d'éléments de type  $i$  doivent être découpés dans le nouveau patron. L'ensemble des variables est donc  $X = (x_1, \dots, x_N)$ . La barre a une taille totale  $L$  et chaque élément a une longueur  $w_i$ . Dans la version la plus simple du problème, l'ensemble des contraintes  $C$  est donc

réduit à une unique contrainte sur la taille maximale utilisable :

$$\sum_{x_i \in X} x_i \cdot w_i \leq L$$

Le réseau de contraintes du problème est donc :

$$(X, C) = ((x_1, \dots, x_N), (\sum_{x_i \in X} x_i \cdot w_i \leq L))$$

Dans les problèmes réels,  $X$  peut contenir d'autres variables comme la couleur ou la taille de la barre utilisée (quand il en existe plusieurs), et  $C$  peut contenir d'autres contraintes comme par exemple la compatibilité entre différents types d'éléments :

$$(x_{i_0} > 0) \Rightarrow (x_{j_0} = 0)$$

**Définition 11** *Le sous-coût  $sc(X)$  est une expression particulière du sous-problème représentant le coût de la colonne correspondant à l'état courant du sous-problème pour le problème maître.*

**Exemple :** Dans le problème de découpe, le coût d'une nouvelle colonne est simplement pris constant. Nous utiliserons :

$$sc(X) = 1$$

**Définition 12** *Le coût réduit  $cr(X)$  est une expression particulière des variables du sous-problème représentant le coût réduit de la colonne correspondant à l'état courant du sous-problème dans le problème maître.*

**Définition 13** *Une contrainte de coût réduit est une contrainte du sous-problème obligeant le coût réduit à être favorable. Pour un problème de minimisation, elle prend la forme :*

$$RCC(X, \alpha) \equiv (cr(X) \leq -\alpha), \text{ avec } \alpha \in \mathbb{R}^+$$

*La contrainte force le coût réduit à être négatif ou positif en fonction du sens de l'objectif du problème maître.*

**Définition 14** *Un objectif de coût réduit est un objectif du sous-problème indiquant que le coût réduit doit être optimisé.*

$$RCO(X, s) \equiv O(cr(X), s)$$

Un sous-problème étant, selon la définition, un problème de satisfaction de contraintes ou d'optimisation, il doit contenir au moins un des deux éléments définis auparavant (une contrainte de coût réduit ou un objectif de coût réduit).

### Variables de Décision

**Définition 15** *L'ensemble des variables de décision est un sous-ensemble  $D$  des variables  $X$  du sous-problème qui sont suffisantes pour construire la colonne associée. Évidemment,  $D \subseteq X$ . Un sous-problème est alors noté :*

$$SP \equiv (X, D, C[\cup RCC(X, \alpha)], [RCO(X, s)])$$

Concrètement, les variables de décision sont celles dont les valeurs sont suffisantes pour calculer les coefficients de la colonne dans le problème maître.

**Exemple :** Dans le problème de découpe, une variable peut être définie pour modéliser la taille de la barre à utiliser. Si la variable n'a pas d'impact sur la colonne générée (i.e. entre autres, le sous-coût ne dépend pas de la taille de la barre), cette variable n'est pas une variable de décision. Les coefficients de la colonne dans le problème maître sont les mêmes quelle que soit la valeur de cette variable.

**Observation 1** *Par définition des variables de décision, le sous-coût, le coût réduit, la contrainte et l'objectif de coût réduit, ne dépendent que des variables de décision. Nous les noterons donc :  $sc(D)$ ,  $cr(D)$ ,  $RCC(D, \alpha)$  and  $RCO(D, s)$ .*

### 3.1.2 Problème Maître

#### Contrainte linéaire

Nous reprenons ici la définition de contrainte linéaire que nous avons déjà précédemment donnée.

**Définition 16** *Une contrainte linéaire  $CL$  est une paire  $(Y, e_{CL}(Y), lb, ub)$  où  $Y$  est un ensemble de variables initiales (éventuellement vide),  $e_{CL}(Y)$  une expression linéaire de ces variables et  $lb$  et  $ub$  des valeurs réelles représentant des bornes inférieure et supérieure pour cette expression.*

Notons que  $lb$  peut être  $-\infty$  et  $ub$  peut être  $+\infty$ . Quand la contrainte utilise un ensemble de variables initialement vide, elle peut être notée  $CL \equiv (lb, ub)$ .

#### Contrainte maître

**Définition 17** *Une contrainte maître  $CM$  pour un sous-problème  $SP$  est un ensemble  $(CL, e_{CM}(D))$  où  $CL$  est une contrainte linéaire et  $e_{CM}(D)$  une expression des variables de décision  $D$  du sous-problème  $SP$ , représentant le coefficient que prendrait une nouvelle colonne dans l'expression linéaire de la contrainte linéaire. L'expression  $e_{CM}(D)$  est appelée l'expression de prix de la contrainte maître  $CM$ .*

**Exemple :** Dans la version simple du problème de découpe à une dimension, le nombre d'éléments de type  $i$  devant être obtenu doit être supérieur à une demande minimale  $d_i$ . Ceci peut être défini par une contrainte maître  $CM_i$  :

$$CM_i = ((d_i, +\infty), x_i)$$

### Objectif linéaire

**Définition 18** Un objectif linéaire  $OL$  est un ensemble  $(Y, e_{OL}(Y), s)$  où  $Y$  est un ensemble de variables (éventuellement vide),  $e_{OL}(Y)$  une expression linéaire de ces variables et  $s$  un sens d'optimisation. Quand l'objectif linéaire utilise un ensemble de variables vide, nous le noterons  $OL \equiv (s)$ .

### Objectif maître

**Définition 19** Un objectif maître  $OM$  pour un sous-problème  $SP$  est une paire  $(OL, e_{OM}(D))$  où  $OL$  est un objectif linéaire et  $e_{OM}(D)$  est une expression des variables de décision  $D$  du sous-problème  $SP$  représentant quel serait le coefficient d'une nouvelle variable dans l'expression linéaire de l'objectif linéaire. L'expression  $e_{OM}(D)$  est ultérieurement appelée expression de prix de l'objectif maître.

**Exemple :** L'objectif maître dans le problème de découpe standard est :

$$OM = ((\text{minimise}), 1)$$

### Problème Maître

**Définition 20** Un problème maître  $PM$  est une paire  $(OM, CM^*)$  où  $OM$  est un objectif maître et  $CM^*$  un ensemble de contraintes maîtres.

### 3.1.3 Problème de Génération de Colonnes

**Définition 21** Un problème de génération de colonnes est une paire  $(PM, SP^+)$  où  $PM$  est un problème maître et  $SP^+$  un ensemble d'au moins un sous-problème.

**Définition 22** Une affectation solution  $A$  pour un problème  $P$  est une affectation  $A \equiv ((x_i, \text{val}_A(x_i)))$ , où  $x_i \in X$  est une variable contrainte et  $\text{val}_A(x_i)$  est une valeur du domaine de  $x_i$  et avec cet ensemble satisfaisant l'ensemble des contraintes du problème et étant optimal par rapport à un éventuel objectif. Pour une affectation solution  $A$ , nous avons pour chaque variable  $x$  une valeur associée  $\text{val}_A(x)$  et pour chaque expression  $e(X)$  une valeur associée  $\text{val}_A(e(X))$  qui est la valeur de l'expression en utilisant pour chacune des variables la valeur de l'affectation.

Quand une nouvelle variable du problème maître  $y_A$  est générée en utilisant l'affectation solution  $A$  d'un des sous-problèmes associés, la colonne associée peut être construite automatiquement en utilisant les valeurs des expressions de prix associées avec l'objectif maître et les contraintes maîtres.

**Observation 2** *Le coefficient dans l'objectif maître du problème maître pour une affectation solution  $A$  d'un des sous-problèmes est  $val_A(e_{OM}(D))$ .*

**Observation 3** *Le coefficient dans la contrainte maître  $CM_i$  du problème maître pour une affectation solution  $A$  d'un des sous-problème est  $val_A(e_{CM_i}(D))$ .*

Ces deux théorèmes sont des résultats directs des définitions des expressions de prix.

En conclusion, pour un objectif maître  $OM$  ou une contrainte maître  $CM_i$ , le coefficient pour une nouvelle colonne est la valeur de l'expression de prix associée.

Donc le *PLNE* défini implicitement par la génération de colonnes est :

$$\begin{aligned} & \text{minimiser/maximiser } \sum_A val_A(e_{OM}(D)).y_A \\ & lb_i \leq \sum_A val_A(e_{CM_i}(D)).y_A \leq ub_i, \forall CM_i \end{aligned}$$

**Observation 4** *Dans un problème de génération de colonnes, l'expression de prix de l'objectif maître est le sous-coût du sous-problème, qui est le même pour tous les sous-problèmes.*

$$e_{OM}(D) = sc(D)$$

### Cas particulier des *expressions de prix* linéaires

**Définition 23** *Une expression des variables de décision est dite être une expression linéaire des variables de décision si et seulement si elle a la forme :*

$$e(D) = \alpha + \sum_{d_i \in D} \beta_i \cdot d_i, \text{ avec } \alpha, \beta_i \in \mathbb{R}$$

Considérons le cas où les expressions de prix  $e_{OM}(D)$  et  $e_{CM_i}(D)$  sont des expressions linéaires des variables de décisions, i.e. :

$$\begin{aligned} e_{OM}(D) &= c + \sum_{d_i \in D} \lambda_i \cdot d_i, \text{ avec } \lambda_i \in \mathbb{R}, c \in \mathbb{R} \\ e_{CM_j}(D) &= c_j + \sum_{d_i \in D} \lambda_{ij} \cdot d_i, \text{ avec } \lambda_{ij} \in \mathbb{R}, \forall CM_j \end{aligned}$$

**Observation 5** *Si les  $(\pi_i)$  sont les valeurs duales associées aux contraintes maîtres  $(CM_i)$ , et que les expressions de prix associées à l'objectif et aux contraintes maîtres ont la forme précédemment définie, le coût réduit a la forme :*

$$\begin{aligned} cr(D) &= sc(D) - \sum_{d_i \in D, CM_j} \pi_j \cdot \lambda_{ij} \cdot d_i \\ &= c + \sum_{d_i \in D} \lambda_i \cdot d_i - \sum_{d_i \in D, CM_j} \pi_j \cdot \lambda_{ij} \cdot d_i \end{aligned}$$

### Variables d'état

Dans certains problèmes, plusieurs sous-problèmes vont générer des colonnes de différents *types*. Ces colonnes venant de différents sous-problèmes peuvent avoir, dans leur problème respectif, des sous-coûts et des expressions de prix a priori différents. Pour faciliter la modélisation de ce type de problèmes sont introduites les *variables d'état* qui permettent d'utiliser les mêmes expressions dans tous les sous-problèmes.

**Définition 24** *A chaque sous-problème  $SP_l$ , ( $l \in \{1, \dots, L\}$ ), nous associons une variable d'état booléenne  $s_l$ . L'ensemble  $S \equiv (s_k)$  des variables d'état est ajouté à tous les sous-problèmes et les valeurs des variables pour une affectation quelconque  $A_l$  du sous-problème  $SP_l$  sont définies ainsi :*

$$\begin{aligned} \text{val}_{A_l}(s_k) &= 1, & \text{si } k &= l \\ \text{val}_{A_l}(s_k) &= 0, & \text{si } k &\neq l \end{aligned}$$

Quand les variables d'état sont utilisées, le sous-coût, le coût réduit, et les expressions de prix s'expriment toutes en fonction des variables de décision et des variables d'état. Elles sont alors notées  $sc(D, S)$ ,  $cr(D, S)$  and  $e(D, S)$ . Les sous-problèmes sont notés  $SP_k \equiv (X, D, S, C, [O])$ .

**Définition 25** *Une expression de prix est dite expression linéaire des variables de décision et d'état si et seulement si elle a la forme :*

$$\begin{aligned} e(D) &= \alpha \\ &+ \sum_{d_i \in D} \beta_i \cdot d_i \\ &+ \sum_{s_k \in S} \delta_k \cdot s_k \\ &+ \sum_{d_i \in D, s_k \in S} \gamma_{ik} \cdot d_i \cdot s_k, \\ &\text{with } \alpha, \beta_i, \delta_k, \gamma_{ik} \in \mathbb{R} \end{aligned}$$

Ce type d'expression de prix est utilisé dans un grand nombre de modèles de génération de colonnes.

**Exemple :** Imaginons un problème de découpe avec différents types de barres disponibles qui ne différencieraient que par leur taille.  $K$  tailles différentes sont disponibles, chaque barre  $k$  a une longueur totale  $L_k$ . Ceci peut être modélisé en utilisant  $K$  sous-problèmes différents avec des états. Le sous-problème  $SP_k$  serait défini ainsi :

$$\begin{aligned}
SP_k &= (D \cup S, C_k) \\
D &= (x_1, \dots, x_N) \\
S &= (s_1, \dots, s_K) \\
C_k &= \left( \sum_{x_i \in D} x_i \cdot w_i \leq L_k \right)
\end{aligned}$$

Dans le cas où le sous-coût serait directement proportionnel à la longueur de la barre utilisée, l'objectif maître aurait la forme suivante :

$$OM = ((\text{minimise}), \sum_{s_k \in S} s_k \cdot L_k)$$

L'expression de prix utilisée dans  $OM$  est un cas particulier d'expression linéaire des variables de décision et d'état avec  $\alpha = 0$ ,  $\beta_i = 0$ ,  $\delta_k = L_k$  and  $\gamma_{ik} = 0$ .

Ainsi les contraintes maîtres assurant la couverture de la demande en éléments de chaque type seraient :

$$CM_i = ((d_i, \infty), x_i)$$

D'autres contraintes maîtres pourraient être ajoutées limitant le nombre de barres disponibles pour un type particulier :

$$CM_k = ((-\infty, n_k), s_k)$$

**Observation 6** *Avec des expressions linéaires des variables de décision et d'état, le sous-coût  $sc(D, S)$  et le coût-réduit  $cr(D, S)$  sont quadratiques, mais peuvent être simplifiés, dans chaque sous-problème  $SP_k$ , par une expression linéaire  $sc_k(D)$  et un coût réduit linéaire  $rc_k(D)$  spécifique au sous-problème.*

### 3.1.4 Coupes

Comme nous l'avons montré dans la section 2.6, il est souvent intéressant de compléter le processus de génération de colonnes avec l'ajout de coupes. Plusieurs utilisations différentes des coupes sont possibles. Nous ne nous intéressons ici qu'à leur formalisation et modélisation dans le cadre de la génération de colonnes.

En génération de colonnes, une coupe ne peut être donnée uniquement par la formulation de la contrainte linéaire sur les variables existantes au moment de l'ajout de la coupe. En effet, la formulation en forme de contrainte linéaire ne tenant compte que des variables existantes n'est qu'une projection de la coupe complète sur le sous-espace défini par les variables courantes. Il est donc nécessaire, afin d'avoir une description complète de la coupe, de lui associer un élément permettant de découvrir les parties manquantes lors de la génération

de nouvelles colonnes. Cet élément peut prendre la forme, tout comme pour les contraintes initialement présentes dans le problème maître, d'une expression de prix. La valeur de cette expression représente le coefficient associé à une éventuelle nouvelle variable. Le processus consistant à donner un coefficient à une nouvelle variable pour une contrainte donnée est connu sous le nom de *lifting*.

Prenons un exemple dans le cadre d'un problème de tournées de véhicules. Les coupes dites de *k-path* sont basées sur la découverte d'un nombre minimal de véhicules nécessaires afin de couvrir un sous-ensemble particulier de visites. Nous ne rentrerons pas dans le détail des techniques utilisées afin de découvrir cet ensemble de visites et ce nombre de véhicules minimum. Notons simplement que étant donné un ensemble de visites  $S$  et  $\kappa(S)$  le nombre minimum de véhicules nécessaire pour couvrir cet ensemble de visites, la coupe peut être donnée par :

$$((\kappa(S), \infty), f_S())$$

L'expression de prix  $f_S()$  est alors une fonction associant à une colonne la valeur 1 ou 0 suivant si la route correspondant à la colonne passe ou non dans cet ensemble  $S$  de visites.

Finalement, tout comme pour la programmation linéaire où toutes les colonnes sont initialement disponibles, le formalisme des expressions de prix permet de rendre équivalentes les descriptions des contraintes présentes initialement et celles (éventuellement redondantes) ajoutées durant la recherche (coupes). Les interactions avec les sous-problèmes sont alors identiques dans les deux cas.

### 3.1.5 Branchement

La création de nouvelles branches correspond à la séparation du problème relâché courant en au moins deux sous-problèmes dont l'union recouvre l'ensemble des solutions entières mais ne contient pas la solution relâchée courante. En général, les systèmes de programmation linéaire offrent deux types de branchement dont la première est un cas particulier de la deuxième.

#### Branchement sur les bornes d'une variable

Étant donnée une variable  $x$  dont la valeur  $x^*$  dans la solution relâchée n'est pas entière, la séparation consiste à fixer dans une branche la borne supérieure de  $x$  à  $\lfloor x^* \rfloor$  et dans l'autre branche la borne inférieure de  $x$  à  $\lceil x^* \rceil$ .

#### Branchement utilisant des coupes

Chacune des branches définies auparavant peut l'être en utilisant deux coupes correspondant aux définitions de bornes précédentes :

$$x \geq \lceil x^* \rceil$$

et

$$x \leq \lfloor x^* \rfloor$$

Ceci peut être généralisé à n'importe quelle coupe. Une correcte utilisation doit assurer :

- que chacune des coupes réduit effectivement l'espace de recherche en éliminant la solution relâchée du nœud père (afin que la recherche puisse aboutir),
- que l'ensemble des coupes n'élimine pas de solution entière (afin de ne pas exclure de solutions).

### Cas de la génération de colonnes

Le cas précédent, générique des deux cas de la programmation linéaire, peut être généralisé à la génération de colonnes en utilisant les expressions de prix. En effet, il s'agit d'utiliser également une partition du problème réalisée en utilisant plusieurs coupes.

Insistons cependant sur un cas particulier qui correspondrait au cas de deux coupes :

$$((1, \infty), f_S()) \text{ et } ((1, \infty), f_T())$$

où  $S$  et  $T$  sont deux sous-ensembles des colonnes du problème et  $f_S()$  (resp.  $f_T()$ ) indique si une colonne appartient ou non au sous-ensemble de colonnes  $S$  (resp.  $T$ ). La partition est donc ici donnée directement en terme de colonnes. Ce cas est fréquent en génération de colonnes. Pour que la génération de colonnes puisse continuer dans chacune des branches correspondantes, il convient que l'algorithme utilisé pour générer des colonnes de coût réduit favorable puisse être modifié afin de ne générer que des colonnes appartenant à ce sous-ensemble  $S$ . Nous dirons alors que le schéma de branchement et l'algorithme de génération sont compatibles. Ceci peut parfois poser problème, et il est important de toujours vérifier ce fait avant d'effectuer un choix de règles de branchement.

Un exemple connu illustre cette possibilité dans le cas des problèmes de tournées de véhicules. Les algorithmes utilisés pour générer des colonnes pour ces problèmes sont en général des algorithmes de programmation dynamique à base d'étiquettes. Une solution au problème relâché peut contenir deux tournées (de valeur non entière) correspondant au même véhicules. Ces deux visites ayant pour même origine le nœud de départ, il sera possible de trouver deux visites  $i$  et  $j$  telles que  $i$  appartient aux deux tournées et où  $j$  est le successeur de  $i$  dans une des tournées et pas dans l'autre. Il est alors possible d'effectuer un branchement similaire à celui précédemment décrit en utilisant comme ensembles  $S$  et

$T$  les ensembles de tournées où le successeur direct de  $i$  est  $j$  (ou bien  $i$  n'est pas présent dans la tournée) (ensemble  $S$ ) et où le successeur direct de  $i$  n'est pas  $j$  (ensemble  $T$ ). Ces deux ensembles forment bien une partition des colonnes correspondant au véhicule donné, et dans chacune des branches, une des deux colonnes non entières est éliminée. Chaque ensemble peut alors être défini par une fonction  $f()$ . Les colonnes appartenant à l'ensemble sont celles dont la valeur pour  $f()$  est 1. Celles dont la valeur est 0 ne sont pas dans l'ensemble. Nous avons alors :

$$f_S() = f_{i \rightarrow j}() = \left( \begin{array}{l} 1 \text{ si } ( \text{ ne contient pas } i) \text{ ou } (j \text{ est successeur direct de } i) \\ 0 \text{ sinon} \end{array} \right)$$

Les fonctions  $f()$  utilisées peuvent en fait être assimilée à des contraintes sur les colonnes. En effet, elles prennent les valeurs 0 ou 1 et s'appliquent aux valeurs définissant la colonne. Dans le cas où un formalisme est utilisé pour décrire le sous-problème de génération de nouvelles colonnes, ce type de branchement pourra alors s'écrire sous forme de contraintes définies par leur fonction de vérité  $f()$ .

Quand une coupe utilisée pour effectuer le branchement a la forme :

$$((0, 0), f_S()),$$

une implantation efficace consiste à déplacer cette fonction  $f_S()$  aux sous-problèmes dans lesquels elle prend la forme d'une contrainte.

## 3.2 Exemples de modèles

En utilisant différents exemples simples, utilisant des sous-problèmes de plus court chemin ou non, nous montrons comment le formalisme précédent permet de définir simplement des problèmes très différents. En effet, ce formalisme permet de cacher les différentes séparations et d'automatiser les interactions entre problème maître et sous-problèmes. Nos applications de la partie III viendront compléter cette série d'exemples.

### 3.2.1 Le problème de découpe à une dimension

Nous utilisons ici le problème de découpe à une dimension tel qu'il est défini dans la section 1.3.1. Reprenons les notations alors utilisées dans la décomposition du modèle en un modèle de génération de colonnes, et en particulier :

- $x_i$  représente le nombre d'éléments  $i$  pris dans le patron généré,

- il existe  $N$  types d'éléments de tailles respectives  $l_i$ , et de demande minimum  $d_i$ ,
- les barres ont une taille maximale  $L$ ,

Le modèle de génération de colonnes donné dans la section 1.3.1 peut alors être formulé ainsi :

$$\begin{aligned}
 PGC &= (PM, SP) \\
 SP &= (D, C, (\text{minimise}, cr(D))) \\
 D &= (x_1, \dots, x_N), x_i \in \mathbb{N} \\
 C &= \left( \sum_{i=1}^N l_i x_i \leq L \right) \\
 sc(D) &= 1 \\
 PM &= (OM, CM_i) \\
 OM &= (\text{minimise}, sc(D)) \\
 CM_i &= ((d_i, \infty), x_i)
 \end{aligned}$$

Nous pouvons déduire de cette description la forme du coût réduit :

$$cr(D) = 1 - \sum_{x_i \in D} \pi_i \cdot x_i$$

### 3.2.2 Crew Scheduling Problem

Le problème connu sous l'acronyme anglais de *Crew Scheduling Problem* peut être vu comme un problème de découpe avec des contraintes d'incompatibilité.  $N$  activités doivent être affectées à des ressources. Une durée de travail maximum  $T$  est donnée et égale pour toutes les ressources. Chaque activité, correspondant à un élément dans le problème de découpe, possède une date de début et de fin, et donc une durée  $d_i$ . Une ressource ne peut donc pas effectuer deux activités qui se recouvrent dans le temps. L'objectif est de trouver un ensemble d'affectations des activités aux ressources, telles que chaque activité est réalisée exactement une fois et qui utilise le nombre minimal de ressources. Nous noterons  $I$  l'ensemble des paires d'activités incompatibles car se recouvrant dans le temps. Le modèle peut alors être écrit :

$$\begin{aligned}
PGC &= (PM, SP) \\
SP &= (D, C, (\text{minimise}, cr(D))) \\
D &= (x_1, \dots, x_N), \text{ avec } x_i \in (0, 1) \\
C &= \left( \begin{array}{l} \sum_{x_i \in D} x_i \cdot d_i \leq T \\ x_i + x_j \leq 1, \forall (i, j) \in I \end{array} \right) \\
sc(D) &= 1 \\
PM &= (OM, CM_i) \\
OM &= (\text{minimise}, sc(D)) \\
CM_i &= ((1, 1), x_i)
\end{aligned}$$

Nous pouvons déduire de cette description la forme du coût réduit :

$$cr(D) = 1 - \sum_{x_i \in D} \pi_i \cdot x_i$$

### 3.2.3 Crew Scheduling Problem avec équivalences

Imaginons que notre problème de *Crew Scheduling* contienne des activités équivalentes. Parmi l'ensemble des  $N$  activités, nous avons  $M$  sous-ensembles  $E_m$  d'activités équivalentes. L'objectif est alors d'effectuer une activité de chaque sous-ensemble exactement une fois. Le modèle devient :

$$\begin{aligned}
C' &= C \cup \left( \sum_{i \in E_m} x_i \leq 1, \forall m \right) \\
PM' &= (OM, CM_m) \\
CM_m &= \left( (1, 1), \sum_{i \in E_m} x_i \right)
\end{aligned}$$

Les nouvelles contraintes du sous-problème signifient qu'il est inutile pour une ressource d'effectuer plusieurs activités équivalentes. Le coût réduit est alors :

$$cr(D) = 1 - \sum_m \pi_m \cdot \left( \sum_{i \in E_m} x_i \right)$$

### 3.2.4 Problème d'Affectation Généralisé

Ce problème, connu sous l'acronyme anglais GAP (*Generalized Assignment Problem*), peut être vu comme un problème de Crew Scheduling où les ressources ne sont pas identiques. Nous avons alors toujours  $N$  activités à affecter aux ressources, mais ces dernières sont limitées au nombre de  $K$  et chacune possède une distribution différente de durées  $d_{i,k}$ , une capacité totale différente  $T_k$  et des gains pour chaque activité  $c_{i,k}$ . Il n'y a plus de notion de superposition des activités dans le temps et il s'agit maintenant de maximiser le gain total. Un modèle peut être :

$$\begin{aligned}
 PGC &= (PM, SP_k) \\
 SP_k &= (D, S, C_k, (\text{maximise}, cr(D, S))) \\
 D &= (x_1, \dots, x_N) \text{ avec } x_i \in (0, 1) \\
 C_k &= \left( \sum_{x_i \in D} x_i \cdot d_{i,k} \leq T_k \right) \\
 sc(D, S) &= \sum_{x_i \in D, s_k \in S} c_{i,k} \cdot x_i \cdot s_k \\
 PM &= (OM, CM_i) \\
 OM &= (\text{maximise}, sc(D, S)) \\
 CM_i &= ((1, 1), x_i)
 \end{aligned}$$

Le coût réduit est alors :

$$cr(D, s) = \sum_{x_i \in D, s_k \in S} c_{i,k} \cdot x_i \cdot s_k - \sum_{x_i \in D} \pi_i \cdot x_i$$

Qui peut être réduit dans chaque sous-problème correspondant à une ressource particulière à :

$$\begin{aligned}
 sc_k(D) &= \sum_{x_i \in D} c_{i,k} \cdot x_i \\
 rc_k(D) &= \sum_{x_i \in D} c_{i,k} \cdot x_i - \sum_{x_i \in D} \pi_i \cdot x_i
 \end{aligned}$$

## Conclusions

Nous avons présenté dans ce chapitre un formalisme permettant une description générique et uniforme des problèmes traitables par génération de colonnes. Les colonnes peuvent alors être automatiquement générées à partir d'une solution

d'un sous-problème et réciproquement, un sous-problème (et en particulier son coût réduit) peut être modifié en fonction de l'état du problème maître. Dans la partie suivante, où nous présentons des applications réelles, nous utiliserons ce formalisme pour présenter d'autres modèles de génération de colonnes.

Auparavant, dans le prochain chapitre, nous proposons une généralisation similaire pour la recherche de solutions en utilisant un formalisme basé sur le concept de *goals* déjà utilisé dans la programmation par contraintes.

# Chapitre 4

## Recherche générique

Une des raisons qui, selon nous, a contribué au retour au premier plan des méthodes de génération de colonnes est l'utilisation de *recherches arborescentes adaptées au problème*.

Nous ne nous référons pas ici à la simple introduction de la méthode de *Branch-and-Price* qui permettrait a priori d'obtenir des solutions optimales ou du moins des bornes intéressantes pour la solution optimale. En effet, il n'est que rarement possible dans les applications réelles d'effectuer une génération complète ou même de parcourir une proportion suffisante de l'arbre de recherche. Souvent, il n'est même pas possible d'accéder aux feuilles de l'arbre donnant des solutions entières dans des temps raisonnables par rapport au besoin industriel. Dans ces derniers cas, qui sont les plus fréquents, l'utilisation de procédures adaptées au problème utilisant des générateurs, règles de branchement et heuristiques de recherche spécifiquement définies nous paraît être un élément déterminant. Comme nous verrons dans le chapitre 7, seule une juste combinaison de ces aspects permet d'accéder à des solutions de qualité dans des temps acceptables pour des applications réelles.

Dans les environnements de génération de colonnes existants, la modification de la procédure de recherche est limitée à la ré-écriture de certains blocs de la procédure : la génération, le branchement, etc. Il est cependant parfois nécessaire de redéfinir la procédure globale elle-même.

Le même type de situation existe dans le cas de la programmation par contraintes pour lequel a été développé le formalisme de *goal* que nous appliquons ici à la génération de colonnes et qui permet une plus grande flexibilité. Nous introduisons tout d'abord la notion de goal, puis montrons comment elle s'applique à la PLNE et à la génération de colonnes. Nous donnons finalement de nombreux exemples d'utilisations pour mettre en œuvre des stratégies spécifiques.

Ce formalisme est disponible dans l'implantation que nous décrivons dans l'annexe A.

## 4.1 Utilisations antérieures des *goals*

Le concept de *goals* n'est pas nouveau. Ceux-ci sont utilisés pour décrire des procédures de recherche dans d'autres domaines.

### 4.1.1 Les Goals en programmation par contraintes

La description de la recherche utilisant le concept de *goal* est présent dans le système de programmation par contraintes ILOG Solver ([ILO02c]) depuis longtemps déjà. Nous présentons ici une simple formalisation de son fonctionnement. Nous donnons également un exemple très simple de fonctionnement.

#### Goal

**Définition 26** *Un goal est un ensemble d'actions devant être réalisées. Quand un goal est exécuté, il effectue une série d'actions et peut ensuite :*

- échouer; alors un retour en arrière au dernier point de choix a lieu. S'il ne reste aucun point de choix, la recherche finit sur un échec.
- retourner un ensemble de sous-goals (éventuellement vide) devant être exécutés à sa place.

**Notation 1** *Une conjonction de goals peut être directement définie en utilisant  $ET(g_1, \dots, g_n)$ . Quand la conjonction est exécutée, chacun des goals  $g_i$  de la liste est exécuté séquentiellement.*

L'exécution des goals est contrôlée via une *pile*. Quand la recherche commence, tous les goals sont ajoutés sur la pile. Un goal est ensuite retiré du dessus de la pile et exécuté. Quand un goal réussit, ses éventuels sous-goals sont mis au dessus de la pile. Nous retournons ensuite à la phase consistant à prendre le goal au dessus de la pile et à l'exécuter. Notons que les sous-goals d'un goal sont donc exécutés avant d'autres goals qui étaient déjà présents dans la pile. La recherche s'arrête quand la pile est vide.

**Notation 2** *Un goal sera introduit en donnant sa procédure d'exécution. Cette procédure peut échouer en appelant la procédure `fail()`, retourner un ou plusieurs sous goals en utilisant `return` ou finir en ne retournant rien avec `return 0`. La notation  $A() = B$  sera utilisée comme équivalent à  $A() = ( \text{return } B )$ .*

#### Point de choix

**Définition 27** *Un point de choix représente un choix entre différents sous goals.*

Chaque sous-goal du point de choix définit une branche dont il sera le dessus de pile. Chaque branche hérite également des goals restant à exécuter. Quand un point de choix est exécuté, la branche définie par le premier choix est exécutée jusqu'à un échec. La pile de cette branche contient donc initialement tous les goals restant lors de l'exécution du point de choix, au dessus desquels est ajouté le sous-goal définissant la branche. En cas d'échec, l'autre branche est ensuite essayée.

**Notation 3** *Un point de choix est défini en utilisant  $OU(g_1, \dots, g_n)$ .*

### Exemple

En utilisant les définitions des goals :

```
A() = ( return 0 )
B() = ( fail() )
C() = ( return ET(A,B) )
D() = ( return OU(B,C) )
```

L'exécution du goal D donnera comme résultat l'exécution successive de :

D, B, echec, C, A, B, echec

Le retour en arrière du premier échec correspond au point de choix mis dans l'appel à D, et la prochaine branche à être essayée est celle définie par C. Quand le second échec a lieu, il n'y a plus de point de choix sur lequel revenir en arrière, la recherche s'arrête sur un échec.

Dans ce cas simple, il serait possible de dessiner un arbre de recherche défini complètement à l'avance, car aucune action complexe non déterminée à l'avance n'est réalisée à l'intérieur des goals. Ceci ne sera plus le cas lorsque les décisions prises dans un goal dépendront de fonctions plus complexes. Dans le cas de la programmation par contraintes, il est fréquent de réaliser des actions en fonction du domaine d'une ou plusieurs variables contraintes. Un autre exemple sera donné dans le cas de la programmation linéaire en nombres entiers dans la section suivante. Dans de tels cas, les goals permettent de définir des heuristiques qui ne sont pas complètement déterminables à l'avance car elles peuvent dépendre à chaque moment de l'état courant de la recherche.

### Optimisation

De manière à optimiser un critère, plusieurs recherches successives sont effectuées, chacune commençant là où la précédente trouve une solution. Le critère à optimiser est contraint à chaque exécution à être meilleur que la dernière solution trouvée. Quand aucune solution nouvelle n'est trouvée, la dernière solution obtenue correspond à la solution optimale pour l'ensemble de la recherche.

### 4.1.2 Les Goals dans les Problèmes Linéaires en Nombres Entiers

Nous donnons un bref exemple d'utilisation de goals dans la PLNE tel que cela est fait avec la dernière version de ILOG CPLEX [ILO02a], ainsi que dans ILOG Hybrid [ILO02b].

#### Recherche traditionnelle de la Problèmes Linéaires en Nombres Entiers

Dans ces deux environnements, des goals peuvent être utilisés pour définir une recherche afin de parcourir l'arbre de recherche. Prenons par exemple le cas d'une variable de type entier  $x$  dont la valeur dans la solution relâchée est fractionnelle. La méthode de branchement habituellement utilisée peut être écrite en utilisant le goal `Branche`.

```
Branche(x) = (
    double val = getCurrentValue(x);
    return OU(x <= floor(val), x >= ceil(val));
)
```

Dans chaque branche définie par l'appel à `OU`, la contrainte respective sera ajoutée qui éliminera la solution fractionnelle courante. De même, l'ensemble de la recherche appliquée à un ensemble de variables de type entier `vars` peut être définie par :

```
Branche(vars) = (
    int index = MostFractional(vars)
    if (index == -1)
        return 0
    return ET(Branche(vars[index]),
              this)
)
```

`MostFractional` est alors une fonction donnant l'index de la variable dont la valeur courante a la partie fractionnelle la plus grande .

#### Recherche personnalisée

Dans la recherche précédemment définie, chacune des branches est définie simplement par la donnée d'une nouvelle borne sur une des variables. Ce type d'heuristique est en général le seul permis par des systèmes dont la recherche ne peut être définie qu'avec des paramètres. Cependant, l'utilisation de goals permet

des formes de branchements beaucoup plus complexes et donc plus adaptés au problème traité.

Par exemple, n'importe quel hyperplan peut être utilisé pour définir les deux branches :

```
Branche(x, y, z) = (
  expr e = x + 2*y + 3*z;
  return OU(e <= 0, e >= 1);
)
```

Bien entendu, ce type de branchement s'avère utile si pour la solution relâchée courante, l'expression utilisée a une valeur strictement comprise entre 0 et 1.

Finalement, tout est possible. Par exemple, alors que jusque là nous utilisions des branchements qui ne perdent pas de solutions et conservent donc la propriété de complétude, il est possible de procéder heuristiquement et d'ajouter des coupes en ne continuant qu'une seule branche ou d'utiliser plusieurs branches ne formant pas une partition.

Par exemple :

```
Branche(x, y, z) = (
  return OR( x - y <= 0, x + y >= 0);
)
```

Notons ici que ceci peut être une manière très intéressante d'utiliser de l'information d'une manière différente qu'en l'ajoutant au modèle, ceci étant d'autant plus intéressant si la partie ajoutée au modèle n'admet qu'une relaxation très mauvaise. Par exemple, une contrainte non linéaire ne doit pas nécessairement être ajoutée sous forme d'une linéarisation, mais peut être traitée en utilisant une série de goals. Ce traitement est pratique pour les contraintes SOS (Special Ordered Set, i.e. contrainte de type cardinalité).

Toutes les combinaisons des goals de base OU, ET et du goal correspondant à l'ajout d'une contrainte peuvent avoir des applications.

## 4.2 Les *goals* en génération de colonnes

Dans cette section, nous présentons notre contribution consistant à utiliser le formalisme de *goals* pour décrire de manière générique les procédures de recherche appliquées à la génération de colonnes. Pour cela, certains *goals* doivent être tout d'abord définis qui pourront ensuite être *assemblés*.

### 4.2.1 Goals pré-définis pour la génération de colonnes

Cet ensemble de goals inclut bien entendu ceux de la PLNE. Nous présentons ici d'autres goals pré-définis pour la génération de colonnes et de coupes.

#### Ajout/Retrait de colonnes

Des colonnes peuvent être ajoutées au problème courant en utilisant explicitement un goal pré-défini recevant un tableau de colonnes :

`Add(cols)`

A l'inverse, des colonnes peuvent être retirées du problème courant en utilisant un goal :

`Remove(cols)`

#### Ajout/Retrait de coupes

De la même manière que pour les colonnes, il est possible d'ajouter et de retirer des coupes au problème courant en utilisant un goal.

`Add(coupes)`

`Remove(coupes)`

#### Élimination de variables en fonction du coût réduit

Le goal `RCClean(val)` élimine du nœud courant toutes les colonnes dont le coût réduit actuel est plus mauvais que la valeur donnée en paramètre. Ce goal est une représentation simple de la méthode proposée dans la section 2.5.1.

#### Réduction par coût réduit et bornes maîtres

Le goal `FixAndSet()` effectue automatiquement des changements de bornes sur les variables tel que cela est proposé dans la section 2.5.1.

#### Utilisation de la PLNE

Un algorithme de programmation linéaire en nombres entiers peut être utilisé sur les colonnes présentes à un nœud de la recherche. Toutes les colonnes n'étant pas disponibles, le résultat ne peut être utilisé pour justifier l'inexistence de solution dans cette branche, mais, au contraire, une solution trouvée forme une borne supérieure pour cette branche.

Le goal `SolveLocalMIP()` applique la recherche en nombres entiers sur les colonnes présentes localement dans le nœud courant. Les coupes et règles de

branchement ajoutées dans la branche et ayant une projection dans le problème maître sont utilisées.

Le goal `SolveGlobalMIP()` effectue la même recherche, mais en utilisant l'ensemble des colonnes générées à tous les nœuds jusqu'alors explorés. Les coupes ajoutées à d'autres nœuds qu'au nœud racine sont retirées ainsi que les règles de branchement.

Ces deux goals sont en particulier intéressants pour essayer de trouver rapidement des solutions (même de qualité moyenne) qui nous permettront ensuite d'éliminer des nœuds et des colonnes. La recherche avec la programmation en nombres entiers est plus rapide pour deux raisons. Théoriquement, certaines opérations ne sont permises qu'en supposant que toutes les colonnes sont connues. C'est le cas par exemple de l'utilisation de certaines coupes qui supposent la présence de toutes les variables. D'autre part, les systèmes de résolution de problèmes linéaires en nombres entiers sont dans la pratique plus efficaces car ils disposent d'années d'expériences et incluent de nombreuses améliorations qui seraient longues à étendre aux cas de génération de colonnes. Citons par exemple l'existence d'heuristiques primales dans le MIP de CPLEX.

### Goal utilisateur

Finalement, tous ces goals pré-définis sont combinés au niveau d'un goal utilisateur. Celui-ci contient un corps de programme effectuant des opérations, prenant des décisions et retournant des sous-goals en fonction de ces décisions. Avant chaque entrée dans un goal utilisateur, le problème maître est actualisé en fonction des actions réalisées auparavant et le simplex est exécuté si le problème linéaire a été modifié. Les informations pouvant en être obtenues, sur lesquelles nous reviendrons dans 4.2.4, correspondent alors à ce problème actualisé.

## 4.2.2 Branchement

Le branchement est réalisé, tout comme en programmation par contraintes, en posant un point de choix en utilisant le goal `OU()`. Dans chacune des branches, des contraintes peuvent être appliquées tant au problème maître (ajout de coupes) qu'aux sous-problèmes (ajout de règles de branchement telles que définies en 3.1.5). Nous pourrions par exemple effectuer le branchement :

```
Branche(regles) = (
  return OU(regles[0], regles[1]);
)
```

L'exécution de ce goal créera deux branches. Dans la première sera ajoutée la règle *rule1* et dans la deuxième la règle *rule2*. Les goals exécutés dans chacune

des branches auront accès à l'ensemble des règles ajoutées jusque là et pourront modifier leurs sous-problèmes en conséquence.

### Stratégies de branchement

L'intérêt principal de l'utilisation de goals est la flexibilité. En effet, le goal effectuant le branchement est libre d'utiliser différents types de règles et de coupes en fonction de différents critères. Par exemple, il est possible de brancher en utilisant des coupes dans les premiers niveaux de l'arbre de recherche, puis d'utiliser des règles de branchement. C'est ce que nous ferons dans le cas de la recherche de solutions heuristiques au problème de conception de réseau (chapitre 7). Par exemple :

```

Branche() = (
  rules = find_rules(getDepth())
  return Branche(rules)
)

```

### 4.2.3 Stratégies de Recherche

Le traitement de chacun des nœuds de l'arbre de recherche étant défini, l'ordre dans lequel ces nœuds seront explorés peut également être modifié. En programmation linéaire, deux principaux types de stratégies de recherche sont utilisées : la recherche en *profondeur d'abord* et en *meilleure borne d'abord*. Ces deux méthodes correspondent grossièrement à la recherche de solution ou de preuve d'optimalité. En programmation par contraintes, d'autres stratégies de recherche sont utilisées. Celles-ci ont en général pour objectif d'obtenir rapidement des solutions d'un certain niveau de qualité. Citons le cas d'une des plus connues, le *LDS* (*Limited Discrepancy Search*), qui permet d'effectuer des explorations partielles de l'arbre de recherche en favorisant des zones où l'existence de solutions est considérée plus probable.

Afin de définir des évaluateurs s'appliquant à des goals définissant des sous-arbres, nous utilisons le même formalisme que celui utilisé dans le système de programmation par contraintes ILOG Solver ([ILO02c]).

Par exemple :

```

BFSGoal(goal) = Apply(goal, BFSEvaluator(cost))

```

Tout le sous-arbre défini par le goal donné sera alors exploré en *meilleure borne d'abord* (BFS : Best First Search en anglais). Ce formalisme permet d'utiliser différentes stratégies de recherche dans différentes parties de l'arbre de recherche.

#### 4.2.4 Informations accessibles aux goals

Les goals permettent une définition non déterministe de la recherche car les décisions sont prises au moment de l'exécution des goals. Afin de prendre ces décisions, certaines informations sont disponibles. Citons, de manière non exhaustive :

- la borne et solution primale correspondant à la meilleure solution trouvée jusqu'à présent. Cette borne permet par exemple de faire des réductions sur le coût réduit. La solution permet par exemple de favoriser, lors d'un branchement, la branche la plus similaire à la meilleure solution trouvée jusqu'alors.
- la borne duale locale. Il s'agit de la solution du problème relâché courant. L'ensemble des deux bornes peut permettre de décider d'abandonner l'exploration du nœud courant si par exemple l'écart est suffisamment faible.
- la profondeur du nœud courant.
- l'ensemble des coupes, colonnes et règles de branchement ajoutées depuis le nœud racine jusqu'au nœud courant, ce qui permet de modifier le sous-problème en conséquence.
- le nombre de fois où le nœud courant correspond à la branche de droite d'un point de choix. Ceci permet d'appliquer des stratégies de recherche du type *LDS*.

#### 4.2.5 Quand et comment éliminer un nœud ?

Par défaut, la génération de colonnes effectuée à un nœud de l'arbre de recherche est supposée complète. Quand un point de choix est posé et que deux sous-nœuds doivent être créés, la borne duale courante est comparée avec la solution primale globale. Le nœud peut alors éventuellement être éliminé si sa borne duale est plus mauvaise que la solution primale.

Si la génération de colonnes utilisée n'est pas complète, une branche pourrait être éliminée alors qu'elle aurait pu, avec une génération complète, donner une nouvelle meilleure solution. Ceci peut être accepté comme une réduction heuristique. Sinon, cette gestion automatique des nœuds peut être *désactivée* pour une branche particulière (éventuellement tout l'arbre de recherche) et réalisée manuellement via des goals spécifiques :

```
SetAssumeFullPriced(boolean)
FailNode()
```

Notons que ce problème ne se pose pas en programmation par contraintes ou en programmation linéaire en nombres entiers car la borne évolue toujours dans le même sens, des variables nouvelles ne pouvant pas être créées durant la recherche.

### 4.2.6 Quand considérer une solution entière comme valide ?

De manière quasi symétrique, le problème maître restreint est supposé complet dans le sens où une solution entière valide est supposée valide pour le problème complet. Ceci n'est pas nécessairement le cas si le nombre de contraintes du problème est trop grand et que celles-ci sont ajoutées dynamiquement. Cette fonctionnalité peut être *désactivée* dans une branche particulière (éventuellement tout l'arbre de recherche). Une nouvelle solution valide est alors prise en compte manuellement via d'autres goals spécifiques :

```
SetCompleteModel(boolean)
TryInteger()
```

## 4.3 Exemples pour la génération de colonnes

Dans cette dernière section, nous donnons brièvement quelques exemples de définitions de recherches en utilisant les goals. Comme pour la modélisation, d'autres exemples seront évidemment fournis par les applications des chapitres suivants.

### 4.3.1 Générateur de colonnes

Un exemple schématique est donné par l'exemple `Genere` :

```
Genere() = (
  ColumnArray cols;
  double[] duals = getDual();
  trouveNouvellesColonnes(cols, duals);

  if (cols.getSize())
    return ET(Add(cols), this);

  return 0;
)
```

Ce goal peut être décrit de la manière suivante. Un tableau initialement vide de colonnes est créé et les valeurs duales des contraintes maîtres sont obtenues. Une fonction est appelée qui se charge d'ajouter dans le tableau des colonnes de coût réduit favorable. Cette fonction pourrait être intégralement écrite dans le goal ; nous l'avons ici mise à part pour des raisons de clarté et pour montrer que la manière dont ces colonnes sont recherchées n'a pas d'importance. Finalement,

si des colonnes ont été ajoutées, le sous-goal `ET(Add(cols), this)` est retourné indiquant qu'il faut ajouter les colonnes au problème courant puis exécuter de nouveau le goal de génération. Quand ce goal est exécuté de nouveau, le problème maître restreint aura été actualisé ainsi que tous les résultats lui correspondant (i.e. le problème linéaire aura été résolu). Enfin, si aucune nouvelle colonne n'est trouvée (i.e. le tableau de colonnes est resté vide), aucun sous-goal n'est retourné. D'autres goals restant dans la pile peuvent alors être exécutés réalisant d'autres types de génération de colonnes, de génération de coupes, un branchement, ...

### 4.3.2 Trouver l'optimum relâché

Une procédure pour obtenir l'optimum relâché est définie par un simple goal :

```
Relax() = Genere()
```

Le goal `Genere` doit bien sûr utiliser une méthode de génération complète et se rappeler successivement comme cela est le cas pour le goal `Genere` que nous avons présenté auparavant afin de pouvoir assurer que la solution relâchée obtenue est optimale.

Notons que cette recherche ne pose aucun point de choix. L'arbre de recherche est donc réduit à une simple branche.

### 4.3.3 Génération de colonnes simple

La génération de colonnes *simple* telle que nous l'avons définie auparavant est également très simple à définir et ne contient pas de point de choix. Après avoir obtenu l'optimum relâché, il suffit d'exécuter une recherche de programmation linéaire en nombres entiers normale :

```
GenColSimple() = ET(Genere(),
                    SolveLocalMIP())
```

Rappelons que cette recherche n'est pas complète mais qu'elle peut s'avérer suffisante dans certaines conditions.

### 4.3.4 *Branch-and-Price*

Nous donnons ici un goal définissant une recherche de type *Branch-and-Price*. Il s'agit d'une forme simple de la procédure.

```

BranchAndPrice() = (
    return ET(Genere(),
             Branche(),
             this))
)

```

### 4.3.5 *Branch-and-Price* avec recherche PLNE sur certains nœuds

Une recherche PLNE peut être exécutée lorsque certains critères sont respectés suggérant qu'une nouvelle solution primale peut être trouvée avec le problème courant. Ces conditions peuvent porter sur le nombre de variables fractionnelles, la valeur de la borne duale, etc. Nous utiliserons ce type de recherche dans le chapitre 7.

```

MyTryInteger() = (
    if (getNbNotInteger() < N)
        return TryInteger()
    else
        return 0
)

```

```

BranchAndPriceInt() = (
    return AND(Genere(),
              MyTryInteger(),
              Branche(),
              this))
)

```

Nous voyons que, entre le *Branch-and-Price* simple et cette version modifiée, le goal principal ne change quasiment pas. De nombreuses autres améliorations peuvent être appliquées en modifiant simplement certains des goals de la recherche.

### 4.3.6 Utilisation de différents générateurs

Comme nous l'avons souligné dans la section 2.5.1, le processus de génération de colonnes peut être accéléré en utilisant différents générateurs, certains étant limités heuristiquement. Nous illustrons ici cette méthode en donnant un goal permettant de réaliser ce type de recherche. Celle-ci est définie en utilisant deux goals `HeurGenere` et `CompGenere` qui utilisent des méthodes de génération respectivement heuristiques et complètes. Le goal de génération heuristique s'exécute

autant de fois que nécessaire jusqu'à ne plus trouver de nouvelles colonnes, alors il retournera le goal de génération complète. Ce dernier quant à lui, retournera le goal de génération heuristique si au moins une colonne est trouvée et ne retournera rien dans le cas contraire.

```
HeurGenere() = (
    ColumnArray cols;
    double[] duals = getDual();
    trouveNouvellesColonnesHeuristique(cols, duals);

    if (cols.getSize())
        return ET(Add(cols), this);

    return ComnpGenere;
)
```

```
CompGenere() = (
    ColumnArray cols;
    double[] duals = getDual();
    trouveNouvellesColonnesComplet(cols, duals);

    if (cols.getSize())
        return ET(Add(cols), HeurGenere);

    return 0;
)
```

Ce type de stratégie pour générer de nouvelles colonnes peut s'avérer extrêmement efficace dans le cas où des sous-problèmes de grande complexité peuvent être réduits heuristiquement de manière significative. Par exemple, dans le cas des problèmes de routage de véhicules, le graphe sur lequel sont recherchées les routes peut être réduit en éliminant certains arcs aléatoirement ou en fonction de certains critères. Quand il n'est plus possible de trouver de colonnes sur ces graphes réduits, le graphe complet est utilisé. Nous utilisons ce type de recherche dans le chapitre 5.

De nombreuses autres situations existent où la possibilité de jouer dynamiquement avec les différents générateurs s'avère extrêmement utile. Par exemple, plusieurs types de règles de branchement peuvent être simultanément utilisées dans le même problème. Si il existe des méthodes efficaces pour résoudre les sous-problèmes où n'interviennent que certains types de règles, un goal de génération peut alors choisir entre différents générateurs en fonction de la liste de règles actives au nœud courant. Nous utilisons ce type de recherche dans le chapitre 7.

### 4.3.7 Génération de colonnes partielle

Certaines règles de branchement peuvent provoquer une simplification importante de la résolution d'un ou plusieurs sous-problèmes. Dans ce cas, il peut être intéressant de ne pas dédier trop de temps à la génération de toutes les colonnes de coût réduit favorables et effectuer rapidement un branchement. Notons que dans ce cas, un soin particulier doit être pris au moment de décider quels sont les nœuds éliminables (comme nous l'avons souligné dans la section 4.2.5). Le sous-problème résultant dans une ou plusieurs sous-branches peut alors être plus simple à résoudre.

### 4.3.8 *Branch-and-Price-and-Cut*

La stratégie de *Branch-and-Price* peut être simplement modifiée pour incorporer une génération de coupes. Nous donnons ici un exemple de goal simple :

```
BranchAndPriceAndCut() = (
    return ET(GenereColonnes(),
              GenereCoupes(),
              Branche(),
              this))
)
```

Comme nous l'avons signalé plusieurs fois déjà, l'ajout d'une colonne (respectivement coupe) peut rendre nécessaire de tester de nouveau les autres coupes (respectivement colonnes) hors problème restreint. Par conséquent, les goals de génération de colonnes ou de coupes devront prendre en compte ce détail et rendre possible le renvoi d'un goal à l'autre. Par exemple, le goal définissant la génération de coupes doit être défini ainsi :

```
GenereCoupes() = (
    CoupeArray coupes;
    double[] valeurs = getValues();
    trouveNouvellesCoupes(coupes, valeurs);

    if (coupes.getSize())
        return ET(Add(coupes), GenereColonnes);

    return 0;
)
```

Ici, nous revenons à la génération de colonnes chaque fois qu'au moins une nouvelle coupe est ajoutée. D'autres combinaisons sont possibles.

## Conclusions

Dans cette section, nous avons montré comment le formalisme de *goals* utilisé déjà pour décrire les procédures de recherche d'autres domaines, peut, sous réserve de prendre certaines précautions, être appliqué à la génération de colonnes.

Nous voyons deux principaux avantages à utiliser les goals pour décrire les procédures de résolution dans le domaine de la génération de colonnes. Le premier est simplement le même que pour les autres utilisations des goals : ceux-ci permettent une description simple de la procédure de recherche. Une procédure de *Branch-and-Price-and-Cut* peut être précisément définie en utilisant quelques instances de goals simples. Le deuxième intérêt repose en partie sur cette simplicité. Dans cette partie, nous avons donné des exemples de procédures de recherche sans préciser leur domaine d'application. En effet, les goals pourront être utiles pour généraliser une procédure efficace pour un problème à d'autres problèmes. Souvent, les mêmes goals pourront être utilisés, seuls les types de colonnes, coupes, et règles de branchement changeront.

Maintenant que les deux formalismes permettant des descriptions génériques des modèles et procédures de recherche pour les méthodes de génération de colonnes sont présentés, nous allons proposer différentes contributions visant à améliorer l'utilisation de ces méthodes pour certains problèmes. Nous utiliserons les formalismes précédents pour décrire nos contributions, ce qui doit faciliter leur utilisation pour d'autres applications.



# Troisième partie

## Applications



Dans cette troisième partie, nous présentons plusieurs améliorations de la méthode appliquée à la résolution de problèmes avec génération de colonnes avec sous-problème de plus court chemin. Nous illustrons ces propositions en utilisant les trois catégories de problèmes que nous avons introduites dans la section 1.4 : le routage de véhicule, la planification de ressources, et la conception de réseau. A chacune de ces familles d'applications est dédié un chapitre qui reprend une description plus formelle du modèle de génération de colonnes utilisant le formalisme de la précédente partie pour un problème particulier de la famille. Une amélioration de la méthode habituellement utilisée est ensuite proposée et son efficacité démontrée à l'aide de résultats concrets sur des instances réputées.

Pour chacune de ces applications, les propositions sont concentrées sur une partie de la procédure de génération de colonnes. La plupart des associations entre amélioration et application sont cependant interchangeables. Les formalismes de la partie précédente permettent une généralisation des résultats au groupe de problèmes global. Certaines combinaisons donneront des résultats intéressants, d'autres non. Par exemple, l'utilisation de chemins élémentaires en planification de ressources est un cas sans intérêt étant donnée la nature intrinsèquement élémentaire des chemins en raison de la ressource temporelle.

Dans la table 4.1 nous donnons un aperçu des combinaisons que nous avons utilisées. Les combinaisons correspondant à "oui" sont utilisées et/ou commentées et avaient déjà été proposées par d'autres auteurs. Celles correspondant à "OUI" sont celles que nous proposons. Dans tous les autres cas, exceptés ceux indiqués par "NON", nos propositions pourraient être utilisées, bien que ces cas ne soient pas illustrés dans nos applications.

	Tournées de Véhicules	Planification de Ressources	Conception de Réseau
Sous-Problème			
Chemins Elémentaires	OUI	NON	
Nouveau schéma de coût		oui	
Géné. Heuristique et Complète	oui	oui	oui
Heuristiques d'expert		OUI	
Stratégies de recherche		OUI	
PPC		oui/OUI	
Problème Maître			
Coupes	oui		OUI
Heuristiques		OUI	OUI
Stratégies de recherche			OUI

TAB. 4.1 – Correspondances entre applications et propositions

Ces trois applications ont été implantées en utilisant l'environnement de génération de colonnes et de coupes que nous avons développé. Cet environnement est composé de deux bibliothèques C++ *Maestro* et *ShortestPath*. La première offre un ensemble de classes C++ permettant de concevoir simplement des applications de génération de colonnes et de coupes en utilisant une modélisation orientée objet. La deuxième offre des facilités pour résoudre les problèmes de plus court chemin apparaissant dans les sous-problèmes. Ces deux bibliothèques sont présentées plus amplement dans l'annexe A.

## Chapitre 5

# Plus court chemin élémentaire : application aux problème de tournées de véhicules

Le point commun entre les différents problèmes présentés dans cette thèse est la nature du sous problème de génération de nouvelles colonnes à coût réduit favorable. Tous comportent un problème de plus court chemin. Ce problème de plus court chemin peut cependant ne pas être toujours exactement identique.

La littérature fait référence à l'utilisation quasi systématique d'un algorithme de plus court chemin avec contraintes de ressources et fenêtres de temps, tel que nous l'avons défini dans la section 2.4.1. Bien que ce modèle s'applique effectivement à un nombre assez large d'applications, il existe des variantes de problèmes pour lesquels il ne peut pas être utilisé directement ou pour lesquels son application directe n'est pas la plus efficace et où une version modifiée peut être utile.

Dans ce chapitre, nous présentons le cas de l'application de la génération de colonnes avec sous-problème de plus court chemin aux problèmes de tournées de véhicules. Nous montrons qu'une version modifiée de l'algorithme permettant de traiter la contrainte d'interdiction des cycles dans les chemins permet d'obtenir des bornes inférieures de meilleure qualité et ainsi de prouver plus efficacement l'optimalité de certaines solutions.

Le chapitre est organisé comme suit. Tout d'abord, dans la section 5.1, nous revenons sur une description formelle du problème de plus court chemin avec contraintes de ressources et fenêtres de temps apparaissant dans le cadre des procédures de génération de colonnes en utilisant les formalismes définis dans les chapitres précédents. Ensuite, nous donnons dans les sections 5.2 et 5.3 une description des problèmes de VRP et discutons de ses possibles extensions. Nous introduisons ensuite notre contribution. Dans la section 5.4 nous discutons de la

validité des modèles avec et sans cycle, dans la section 5.5 nous comparons les bornes inférieures obtenues avec les différents modèles, puis dans la section 5.6 nous présentons des modifications à l'algorithme initial à base d'étiquettes pour prendre en compte cette contrainte et dans 5.7 plusieurs procédures complètes utilisant des réductions heuristiques de cet algorithme. Enfin, les dernières sections présentent des améliorations techniques et des résultats encourageants sur un benchmark reconnu.

Les résultats présentés dans ce chapitre ont été en partie obtenus lors d'une étude plus générale sur la coopération des méthodes de génération de colonnes avec les méthodes de recherche locale réalisé avec Claude Le Pape et Émilie Danna.

## 5.1 Le problème du plus court chemin avec fenêtres de temps pur en génération de colonnes

Dans cette section, nous présentons le problème de plus court chemin avec fenêtres de temps. Les notations utilisées ici reprennent celles de la section 2.4.4 et seront appliquées tant pour les problèmes résolus en utilisant l'algorithme d'origine à base d'étiquettes, que l'algorithme modifié que nous présentons ou encore les résolutions à base de programmation par contraintes du prochain chapitre. Nous utiliserons également les notations suivantes :

- $N$  visites sont à effectuer, le départ et l'arrivée se font à un nœud spécifique appelé dépôt et représenté par deux nœuds spécifiques 0 et  $N + 1$ .
- des variables  $next_i$  représentent le chemin parcouru. Chaque variable  $next_i$  aura pour valeur le nœud suivant dans le chemin. Un nœud non utilisé aura sa variable instantiée à  $i$ .
- Il existe  $L$  ressources. Pour chaque ressource  $l$  existent des variables  $cumul_{l,i}$ . La variable  $cumul_{l,i}$  représente l'accumulation le long du chemin pour la ressource  $l$  du nœud initial jusqu'au nœud  $i$ .
- $pathlength(next, cumul_l, dist_l)$  représente une contrainte sur l'ensemble des variables  $next_i$  et  $cumul_{l,i}$ , pour  $i$  de 0 à  $N + 1$  obligeant ces dernières à respecter l'accumulation de ressources donnée par la distance  $dist_l$

Le problème de ESPRCTW peut donc être décrit de la manière simple suivante :

$$\begin{aligned} pathlength(next, cumul_l, dist_l), & \quad \forall l \in \{1, \dots, L\} \\ min_{l,i} \leq cumul_{l,i} \leq max_{l,i}, & \quad \forall l \in \{1, \dots, L\}, \forall i \in \{1, \dots, N + 1\}. \end{aligned}$$

Souvent, les contraintes maîtres font état d'une couverture d'une demande correspondant à un nœud particulier. Les expressions de prix correspondantes

font alors intervenir une expression  $\delta_i^p$  devant indiquer si le chemin  $p$  passe par la visite  $i$ . Cette expression peut s'écrire simplement :

$$\delta_i^p = (\text{next}_i \neq i).$$

Ce problème de plus court chemin est généralement résolu en utilisant un algorithme de programmation dynamique à base d'étiquettes où la contrainte d'élémentarité des chemins est relâchée.

Dans ce chapitre, nous allons montrer comment la modification de l'algorithme habituel afin de ne pas accepter les cycles dans les chemins permet d'obtenir de meilleurs résultats dans certains cas. En particulier, nous présentons une application de cette modification à l'algorithme de plus court chemin dans le cadre du problème du VRPTW. Nous montrons comment cet algorithme modifié, combiné avec d'autres heuristiques, nous permet d'obtenir et de prouver des solutions optimales sur une quinzaine d'instances ouvertes reconnues. Nous commencerons par donner des présentations formelles complètes des problèmes de VRP et VRPTW, et par donner une idée des extensions possibles de ces problèmes.

## 5.2 Problèmes de VRP et VRPTW

Le problème de VRP, ainsi que les deux problèmes obtenus par décomposition, ont été introduits formellement dans la section 1.3.2. Nous donnons ici un modèle complet en utilisant le formalisme du chapitre 3.

Dans le VRP sont utilisées deux ressources :

- la capacité utilisée par les biens à livrer est limitée par la capacité totale  $C_T$  du camion. Nous noterons  $\text{cumul}_{c,i}$  et  $\text{dist}_c$  les variables et le distancier associés à cette ressource. Le distancier sera simplement défini tel que  $\text{dist}_c(i, j)$  est la capacité utilisée par la livraison  $j$ .
- la longueur parcourue, définissant le coût d'une route. Nous utiliserons les notations  $\text{cumul}_{l,i}$  et  $\text{dist}_l$  pour représenter les variables et le distancier associés à cette ressource. Le distancier est défini tel que  $\text{dist}_l(i, j)$  représente la distance de  $i$  à  $j$ .

Le modèle décomposé en utilisant le formalisme du chapitre 3 est alors :

$$\begin{aligned}
PGC &= (PM, SP) \\
SP &= (X, D, C, O(\text{minimise}, cr(D))) \\
X &= (D \cup \text{cumul}_l \cup \text{cumul}_c) \\
D &= (\text{next}_0, \text{next}_1, \dots, \text{next}_{N+1}) \\
C &= \left( \begin{array}{c} \text{pathlength}(\text{next}, \text{cumul}_l, \text{dist}_l) \\ \text{pathlength}(\text{next}, \text{cumul}_c, \text{dist}_c) \\ \text{cumul}_{c,N+1} \leq C_T \end{array} \right) \\
PM &= (OM, CM_i) \\
OM &= (\text{minimise}, sc(D)) \\
sc(D) &= \text{cumul}_{l,N+1} \\
CM_i &= ((1, 1), \text{next}_i \neq i)
\end{aligned}$$

Notons que pour la résolution pratique, les contraintes  $CM_i$  sont modifiées pour simplement contraindre la couverture des visites :

$$CM_i = ((1, \infty), \text{next}_i \neq i).$$

Il est facile de voir qu'une solution au problème initial est aussi solution pour ces nouvelles contraintes. D'autre part, il est évident que la solution entière optimale du nouveau problème ne contiendra qu'une route contenant chaque visite.

Dans le cas du VRPTW, une dimension supplémentaire, le temps, est introduite. Elle est représentée par les variables d'accumulation  $\text{cumul}_t$ . Le distancier est celui de la ressource de longueur. L'accumulation du temps en chacun des nœuds est limitée par des bornes  $[a_i, b_i]$ . Seul le sous-problème est modifié et devient :

$$\begin{aligned}
SP &= (X, D, C, O(\text{minimise}, cr(D))) \\
X &= (D \cup \text{cumul}_t \cup \text{cumul}_l \cup \text{cumul}_c) \\
D &= (\text{next}_0, \text{next}_1, \dots, \text{next}_{N+1}) \\
C &= \left( \begin{array}{c} \text{pathlength}(\text{next}, \text{cumul}_t, \text{dist}_l) \\ \text{pathlength}(\text{next}, \text{cumul}_l, \text{dist}_l) \\ \text{pathlength}(\text{next}, \text{cumul}_c, \text{dist}_c) \\ \text{cumul}_{c,N+1} \leq C_T \\ a_i \leq \text{cumul}_{t,i} \leq b_i, \forall i \in \{1, \dots, N\} \end{array} \right)
\end{aligned}$$

Cette forme de sous-problème est commune à de nombreux problèmes. Les méthodes proposées ici seront alors facilement réutilisables sur ces autres problèmes. Elles ne seront cependant pas toujours efficaces.

## 5.3 Autres extensions au VRP

D'autres extensions au VRP ont été proposées afin de correspondre au mieux aux problèmes réels.

### 5.3.1 Flottes hétérogènes

Nous imaginons difficilement qu'une entreprise dispose d'une flotte de véhicules uniforme. Dans le meilleur des cas, plusieurs catégories contenant chacune plusieurs véhicules seront disponibles et même peut-être tous les véhicules auront des caractéristiques différentes. Parmi celles utilisées dans le cas du VRPTW, les capacités, schémas de coûts, fenêtres de temps et même les distanciers peuvent être différents dans différents sous-problèmes. Ce dernier cas permet de prendre en compte des vitesses de déplacement différentes. Le modèle avec véhicules non identiques consiste alors simplement à utiliser plusieurs sous-problèmes  $SP_k$  ayant chacun des caractéristiques différentes. Il restera alors, si la situation se présente, à limiter le nombre de véhicules par catégorie en utilisant une contrainte maître  $CM_k$  par catégorie :

$$CM_k = ((0, m_k), s_k)$$

avec  $s_k$  la variable d'état correspondant au sous-problème  $SP_k$ , telle que nous l'avons définie dans la section 3.1.3 et  $m_k$  le nombre maximal de véhicules pour cette catégorie. Dans [Tai99] est présentée une méthode de résolution heuristique pour cette variante.

### 5.3.2 Dépôts multiples

L'existence de dépôts multiples est également fréquent dans la réalité. Les positions de ces dépôts peuvent d'ailleurs être le résultat d'un autre problème d'optimisation connu sous le nom anglais de *Warehouse Location Problem* et pour lequel des méthodes de génération de colonnes peuvent être utilisées. Le sous-problème n'est alors pas de plus court chemin. La même méthode que pour le cas des catégories de véhicules peut être utilisée ici en prenant des distances entre le dépôt et les visites, différentes pour chaque sous-problème. Certaines visites peuvent également n'être accessibles qu'aux véhicules d'un dépôt particulier. Il suffira alors d'éliminer le nœud des graphes des autres sous-problèmes.

## 5.4 Problèmes avec cycles et sans cycles

Les graphes des modèles de plus court chemin utilisés en génération de colonnes ne sont pas tous acycliques par nature. Dans les cas de problème de planification de ressource, les graphes sont acycliques, car chaque nœud représente

une activité ou un travail réalisable, dont la date est déjà fixée. Un plan de travail complet est alors donné par la liste, sans ordre particulier, des activités à réaliser. L'ordre est naturellement et implicitement défini par les horaires des activités. Dans le cas du problème de VRP, la situation est différente, car il s'agit d'affecter les visites à des routes, mais également de choisir l'ordre dans lequel ces visites sont effectuées, cet ordre n'étant pas défini à l'avance. Le graphe sur lequel est cherché le plus court chemin permet donc aux visites d'être placées dans des ordres différents. Au niveau du graphe, ceci correspond, pour deux activités  $i$  et  $j$ , à l'existence des arcs  $(i, j)$  et  $(j, i)$ . Si le premier arc est utilisé, la visite  $j$  est effectuée juste après la visite  $i$  et dans le deuxième cas, la visite  $i$  juste après  $j$ . L'existence de ces deux arcs rend possible la définition d'un chemin dans le graphe incluant des cycles, i.e. où la visite  $i$  ou  $j$  sont présentes plusieurs fois.

### 5.4.1 Cas du VRP

L'algorithme de plus court chemin habituellement utilisé à base d'étiquettes autorise de tels chemins alors que le problème de VRP n'autorise que les chemins élémentaires. Cet algorithme est cependant utilisé pour deux raisons :

- même en acceptant des cycles dans les colonnes générées, il est possible de montrer que la solution entière finale n'en comportera pas,
- de plus, la complexité d'un algorithme acceptant les cycles est à priori meilleure,

### 5.4.2 Validité de l'utilisation de chemins non-élémentaires

Dans [CDP02], nous avons brièvement expliqué comment la contrainte sur le caractère élémentaire des chemins étant relâchée, il est cependant possible d'assurer que la solution optimale entière ne contiendra pas de chemin contenant des cycles.

**Proposition 3** *La solution optimale entière d'un modèle de génération de colonnes au problème du VRP où le problème maître relâché accepte des colonnes contenant des cycles et où les distances respectent l'inégalité triangulaire, ne contiendra que des chemins élémentaires.*

En effet, imaginons que  $S$  soit une solution optimale au problème et contienne un chemin  $p$  contenant un cycle. Il existera une visite  $i$  qui sera effectuée deux fois par ce chemin. Nous pouvons alors construire un chemin  $p^*$  identique au chemin  $p$  mais où une des deux visite  $i$  est supprimée. La solution  $S^*$  construite à partir de la solution  $S$  où  $p$  est remplacé par  $p^*$  est valide et de coût inférieur ou égal. Sa validité est évidente, toutes les visites étant couvertes de la même manière sauf  $i$  qui est toujours couverte au moins une fois. Il est facile de voir que le

coût de  $S^*$  sera égal au coût de  $S$  plus la différence entre les coûts de  $p$  et  $p^*$ . Hors, en utilisant l'hypothèse que la fonction des coûts  $c(i, j)$ , respecte l'inégalité triangulaire, nous voyons que la différence entre les coûts est positive.

$$c(\text{prec}(i), i) + c(i, \text{succ}(i)) - c(\text{prec}(i), \text{succ}(i)) \geq 0$$

ce qui montre que le coût de  $S^*$  est inférieur ou égal au coût de  $S$ . La même hypothèse permet d'assurer la validité du chemin  $p^*$ .

## 5.5 Comparaison des bornes inférieures

A chaque nœud de l'arbre de recherche du *Branch-and-Price*, la valeur optimale du problème maître relâché, quand aucune nouvelle colonne de coût réduit négatif ne peut être générée, constitue une borne inférieure pour le problème en nombres entiers pour la branche correspondante. La qualité de ces bornes est importante car elle permet de réduire la taille de l'arbre en éliminant certains nœuds. Dans cette section, nous présentons deux techniques qui permettent d'améliorer leur qualité. La première consiste à ajouter des coupes à la relaxation et a déjà été largement documentée dans la littérature. La deuxième, qui correspond à notre contribution, est basée sur l'utilisation d'un sous-problème de plus court chemin élémentaire.

Nous noterons PM et PME les problème maîtres correspondant respectivement à l'utilisation du SPRCTW et ESPRCTW pour résoudre les sous-problèmes, et PMR et PMER leur relaxation quand les contraintes d'intégralité sont relâchées.

### 5.5.1 Exemple simple

Reprenons l'exemple de la section 2.2.2 avec des capacités et fenêtres de temps larges. La solution optimale pour PMER est évidemment faite de l'unique route  $d - i - j - d$  avec  $LB_{elem} = 201$ . Quand les cycles sont acceptés (pour PMR), n'importe quelle route de la forme  $d - (i - j)^n - d$ , avec  $i - j$  répété  $n$  fois, peut faire partie d'une solution avec la valeur  $1/n$ , donnant ainsi une borne  $LB = 100/n + n$  où  $n$  n'est limité que par la capacité des véhicules. Une route de ce type est représentée dans la figure 2.1.

### 5.5.2 Utilisation de coupes

La méthode la plus utilisée pour améliorer la solution obtenue avec la relaxation consiste à ajouter des coupes telles que les coupes de  $k - path$ , connues sous le nom de sous-tour lorsque  $k = 1$ .

Soit  $S$  un ensemble de visites,  $\delta(S)$  l'ensemble des arcs sortants de  $S$ , c'est à dire  $\delta(S) = \{(i, j) \in A / i \in S, j \notin S\}$  et  $\mu_S^p$  le nombre d'arcs que le chemin  $p$  possède dans  $\delta(S)$ . En utilisant différentes techniques, il est possible de démontrer que les visites de cet ensemble ne peuvent pas être couvertes en utilisant moins de  $\kappa(S)$  véhicules. Pour tout  $k \leq \kappa(S)$ , une nouvelle contrainte sur les variables  $\lambda$  correspondant aux routes contenant au moins une visite dans cet ensemble peut alors être ajoutée :

$$\sum_{p \in \Omega} \mu_S^p \lambda^p \geq k$$

Il est clair que quelque soit  $S$ , nous aurons toujours  $\kappa(S) \geq 1$ . L'utilisation des coupes de sous-tours (cas où  $k = 1$ ) ne nécessite alors aucune démonstration préalable. Une coupe de sous-tour peut alors être utilisée sur l'exemple précédent, en utilisant l'ensemble  $S = \{i, j\}$  qui aurait un flux de  $1/n$ , et serait donc violée. Ajouter cette coupe améliorerait probablement la qualité de la borne.

Cependant la borne obtenue avec PMR plus les coupes de sous-tours n'est pas équivalente à celle obtenue avec PMER. Il a été démontré dans [Koh95] qu'en utilisant ESPRCTW, la solution ne violait aucune coupe de sous-tours. Le contraire n'est pas vrai.

Illustrons ceci avec un autre exemple. Supposons que nous avons 3 visites  $i, j$  et  $k$ . L'ensemble de routes suivant peut être une solution où chaque route prend la valeur  $\frac{1}{2}$ .

$$\begin{aligned} & d - j - k - d \\ & d - i - j - k - i - d \end{aligned}$$

Même si cette solution ne viole aucune coupe de sous-tour, elle contient un cycle et ne serait donc pas solution de PMER.

Enfin, il est important de se rappeler que même si la résolution de ESPRCTW peut être complexe, la génération de coupes fait également appel à des algorithmes complexes pour isoler les coupes violées par la solution courante et devant être ajoutées à la relaxation. Dans le cas des coupes de  $k - path$ , un algorithme complexe pour trouver les  $\kappa(S)$  doit nécessairement être utilisé.

### 5.5.3 Utilisation de chemins élémentaires

Les contraintes de PM et PME étant identiques et les ensemble de colonnes étant inclus l'un dans l'autre, il est facile de voir que PM est une relaxation de PME et PMR une relaxation de PMER. Nous avons vu auparavant que PM et PME avaient les même solutions entières. Ce résultat n'est pas vrai pour PMR et PMER, mais la relation de relaxation implique que la borne obtenue avec PMER soit de qualité égale ou supérieure que celle obtenue avec PMR. Il peut même

arriver que la solution de PMER soit meilleure et entière. Ceci a été le cas dans la moitié des instances de VRPTW que nous avons fermées. Ceci est clairement illustré dans l'exemple précédent.

Notre idée est donc d'utiliser un algorithme d'ESPRCTW pour les sous-problèmes même si cet algorithme a en théorie une bien plus mauvaise complexité. Nous devons utiliser une implantation qui offre un bon compromis entre une efficacité moyenne d'exécution dans la pratique et une bonne qualité de borne théorique.

## 5.6 Plus-court chemins élémentaires

Dans cette section, nous proposons différentes modifications à l'algorithme à base d'étiquettes pour la résolution de problème de plus court chemin élémentaire.

Dans le cas de chemins élémentaires, deux différences existent :

1. un chemin partiel ne peut pas être continué avec un nœud déjà présent dans le chemin partiel,
2. une étiquette ne peut pas être directement éliminée en utilisant la règle de dominance habituelle.

Le premier élément signifie qu'un chemin partiel finissant en  $i$  peut être prolongé avec le nœud  $j$  si et seulement si  $j$  n'a pas déjà été visité par le chemin partiel. Si nous notons  $V(c)$  l'ensemble des nœuds du chemin partiel  $c$ , alors nous ne pouvons pas prolonger  $c$  à  $j$  si nous avons  $j \in V(c)$ .

Le deuxième élément est la principale difficulté. En effet, même si un chemin partiel  $c_1$  domine un autre chemin partiel  $c_2$ , une continuation de  $c_2$  avec une visite  $i \in V(c_1)$  peut être utile ultérieurement. Nous devons alors conserver  $c_2$  car à  $c_1$  ne peut pas être appliquée la même prolongation comme cela est fait dans la preuve de la règle de dominance. En fait, une règle de dominance facilement modifiée pourrait être appliquée qui consiste à n'éliminer aucune étiquette. Même si ceci amène théoriquement à trouver le chemin optimal, c'est impossible à réaliser dans la pratique, le nombre d'étiquettes augmentant exponentiellement avec le nombre de nœuds. Cette difficulté pratique semble être une de celles qui font que la relaxation au SPRCTW est habituellement utilisée.

Notre objectif est donc d'améliorer cette règle de dominance. La nouvelle règle de dominance devrait permettre d'éliminer autant de chemins partiels que possible avec la sécurité qu'ils ne font pas partie de la solution optimale. Cette règle de dominance doit également être applicable en un temps acceptable.

### 5.6.1 Une première règle de dominance modifiée

La règle de dominance d'origine ne permet pas d'assurer que  $c_2$  ne peut pas être mieux continué que  $c_1$ , car  $c_2$  peut éventuellement être continué avec un nœud déjà visité par  $c_1$  et qui ne peut donc pas être ajouté à  $c_1$ . Une règle de dominance modifiée (déjà proposée dans [DD86]) est donc utilisée où la condition suffisante pour un chemin partiel  $c_1$  de dominer un chemin partiel  $c_2$  est de respecter les conditions (2.1) et (2.2) de la définition d'origine définies dans la section 2.4.2, et d'avoir également son ensemble de visites  $V(c_1)$  inclus dans l'ensemble  $V(c_2)$ .

$$V(c_1) \subseteq V(c_2) \tag{5.1}$$

Évidemment, la nouvelle règle de dominance va conserver de nombreuses étiquettes ultérieurement inutiles. En effet, quand deux chemins ont un ensemble de visites non inclus l'un dans l'autre (et donc la condition (5.1) non respectée), aucune domination ne sera applicable entre les deux étiquettes.

Nous introduisons alors d'autres améliorations à cette règle, tout d'abord une amélioration qui conserve l'ensemble des propriétés de la règle, puis certaines modifications heuristiques mais qui peuvent cependant être utilisées efficacement dans une procédure globalement complète.

### 5.6.2 Amélioration complète

Si le chemin partiel  $c_2$  ne contient pas certaines visites du chemin partiel  $c_1$ , l'étiquette lui correspondant sera systématiquement conservée, même si chacune des valeurs est respectivement plus mauvaise. La raison en est que  $c_2$  pourrait devenir intéressant en passant par l'une de ces visites. Cette condition suffisante n'est bien sûr pas nécessaire. La condition peut alors être améliorée en estimant le gain de coût réduit qui pourrait être obtenu en passant par certaines de ces visites. Notons que cette amélioration ne tient pas compte de la nature réelle du coût réduit et ne constitue pas une méthode à rapprocher de 2.5.1.

Soient deux chemins partiels  $c_1$  et  $c_2$  arrivant au même nœud  $p$ . Rappelons que  $V(c)$  est l'ensemble des nœuds traversés par  $c$ .

Si  $V(c_1) \setminus V(c_2) = \emptyset$  et que  $c_1$  domine  $c_2$  selon la règle simple, il n'y a pas de difficulté particulière. Il s'agit de la nouvelle règle de dominance. Remarquons également qu'un élément  $n$  de  $V(c_1) \setminus V(c_2)$  peut ne pas être considéré s'il ne peut pas être ajouté à  $c_2$ . En effet, s'il existe une ressource  $l$  telle que  $D_2^l + d^l(p, n) > b_n^l$ , il sera impossible d'ajouter  $n$ . Nous utilisons ici la propriété d'inégalité triangulaire.

Considérons maintenant le cas où  $V(c_1) \setminus V(c_2) = \{n\}$  :

1. minoration de l'amélioration avec  $\pi_n$ . Quelle que soit  $c_2^*$  une prolongation du chemin partiel  $c_2$ , il sera possible de prolonger  $c_1$  en  $c_1^*$  de la même manière, sauf pour  $n$ . Si nous notons  $i$  et  $j$  les prédécesseur et successeur de  $n$  dans  $c_2^*$ , nous aurons alors :

$$rc_2^* - rc_1^* = c(i, n) + c(n, j) - c(i, j) - \pi_n + rc_2 - rc_1$$

Donc  $c_2^*$  est meilleur que  $c_1^*$  si et seulement si :

$$c(i, n) + c(n, j) - c(i, j) - \pi_n + rc_2 - rc_1 < 0$$

Or, en raison de l'inégalité triangulaire,  $c(i, n) + c(n, j) - c(i, j) \geq 0$ , une condition suffisante pour éliminer  $c_2$  est donc :

$$-\pi_n \geq rc_1 - rc_2$$

2. minoration de l'amélioration avec  $\pi_n - \min\text{Cout}_n$ . Il s'agit ici de minorer  $c(i, n) + c(n, j) - c(i, j)$  par :

$$\min\text{Cout}_n = \min_{i \in \text{prec}(n), j \in \text{succ}(n)} (c(i, n) + c(n, j) - c(i, j))$$

où  $\text{prec}(n)$  et  $\text{succ}(n)$  sont respectivement l'ensemble des prédécesseurs et des successeurs possibles du nœud  $n$ . Cette quantité peut être calculée une fois pour toutes à l'initialisation de l'algorithme. Le chemin  $c_2$  peut donc être éliminé dès que :

$$\min\text{Cout}_n - \pi_n \geq rc_1 - rc_2$$

Ces modifications sont généralisables au cas où  $\text{Card}(V(c_1) \setminus V(c_2)) > 1$ . Mais dans la pratique, des minorants intéressants sont difficiles à calculer. Nous nous sommes donc restreints au cas où  $\text{Card}(V(c_1) \setminus V(c_2)) \leq 2$ . Bien que plus difficile à calculer, un minorant intéressant sur le coût dans le cas de deux éléments peut également être obtenu dès l'initialisation.

## 5.7 Améliorations heuristiques

Résoudre le problème de plus court chemin élémentaire avec la règle de dominance définie dans 5.6.1 peut être difficile dans la pratique. Même en utilisant les améliorations complètes de la section précédente, la résolution de notre sous-problème peut prendre beaucoup de temps pour ne trouver que quelques colonnes. En fait, au début du traitement d'un nœud, obtenir une nouvelle route de coût réduit négatif peut ne pas nécessiter la complétude de l'algorithme. N'importe quelle route élémentaire de coût réduit négatif peut être utilisée. Nous proposons donc dans cette section diverses modifications heuristiques de l'algorithme précédent, qui seront combinées avec la version complète d'une manière similaire à celle proposée dans la section 4.3.6.

### 5.7.1 Réduction heuristique simple

Une manière simple d'obtenir une version heuristique de notre algorithme consiste à ne tenir compte que du premier élément des différences présentées dans la section 5.6 (i.e. interdire les extensions des routes à des nœuds déjà visités) mais sans modifier la règle de dominance. Seuls des chemins élémentaires seront générés bien qu'une étiquette correspondant à un chemin partiel de la solution optimale sans cycles puisse être éliminée. Quand cet algorithme ne trouve aucun chemin de coût réduit négatif, ceci ne signifie pas qu'un tel chemin n'existe pas car une étiquette partielle a pu être éliminée.

Nous utilisons alors une procédure consistant à utiliser l'algorithme heuristique tant qu'il permet de trouver de nouveaux chemins de coût réduit favorable, puis, quand ceci n'est plus possible, à utiliser la version complète. Si la version complète trouve un nouveau chemin, nous revenons à la version heuristique. Quand la version complète ne trouve pas de chemin, le traitement du nœud aboutit.

### 5.7.2 Niveaux de dominance

Il existe une grande différence pratique entre les deux versions de l'algorithme utilisées dans la procédure précédente. Parfois, la version heuristique finit rapidement sans trouver de nouvelles colonnes, mais l'exécution de la version complète tarderait énormément. Nous introduisons alors un paramètre permettant de définir des versions intermédiaires limitant de manière plus fine la *complétude* de l'algorithme. Le paramètre *DominanceLevel* définit la taille des chemins partiels à partir desquels la règle de dominance complète est appliquée. Si le chemin partiel est plus court, la version heuristique est utilisée. Les deux versions (heuristique et complète) correspondent alors aux deux valeurs extrêmes  $\infty$  et 0 de *DominanceLevel*. La procédure de recherche pour le nœud consiste maintenant à commencer avec une valeur du paramètre très grande, et à la diminuer quand il n'est plus possible d'obtenir de nouvelles colonnes.

### 5.7.3 Réductions heuristiques du graphe

Comme ceci est proposé pour les problèmes de plus court chemin avec cycles (e.g. dans [DDS01]), le graphe des visites peut être réduit heuristiquement afin de réduire le temps nécessaire à sa résolution. Nous éliminons des arcs en fonction de leur *distance* pour une des ressources (nous avons ici utilisé la longueur). Différents critères ont été utilisés tels que :

$$\begin{aligned} d^l(i, j) &\geq \textit{percent} * \max_{k \in \textit{succ}(i)} d^l(i, k) \\ d^l(i, j) &\geq (1 + \textit{percent}) * \min_{k \in \textit{succ}(i)} d^l(i, k) \end{aligned}$$

Comme pour les réductions précédentes, nous augmentons le pourcentage du graphe conservé quand aucun chemin n'est trouvé.

Notons que les trois procédures que nous venons de décrire, bien qu'elles utilisent une version heuristique de l'algorithme, sont globalement complètes, i.e. aucune colonne valide et de coût réduit favorable n'est perdue.

## 5.8 Autres améliorations dans l'implantation

Les sections précédentes ont introduit des améliorations algorithmiques pour la procédure de pricing. Nous présentons ici quelques idées que nous avons utilisées pour l'implanter de manière plus efficace.

### 5.8.1 Mémoire d'étiquettes

Durant les nombreuses exécutions successives de l'algorithme légèrement modifié ou limité, une partie du temps d'exécution est utilisée à démarrer l'algorithme, i.e. recréer beaucoup d'étiquettes présentes à la fin de l'exécution précédente et arriver à un état similaire. Nous utilisons alors une *mémoire d'étiquettes* d'une manière incrémentale. Diverses opérations peuvent être réalisées sur cette mémoire, e.g. la remplir avec les étiquettes à la fin d'une exécution, ou l'utiliser pour initialiser une autre exécution. Les étiquettes venant de la mémoire et ajoutées à un algorithme sont actualisées (les coûts associés sont recalculés) et certaines éventuellement éliminées si elles sont dominées ou ne correspondent pas à un chemin valide pour le modèle modifié.

## 5.9 Autres améliorations

### 5.9.1 Réductions du graphe utilisant les fenêtres de temps

Comme proposé dans [DD86] nous supprimons de l'ensemble des arcs possibles ceux que les fenêtres de temps aux nœuds de départ et d'arrivée rendent impossibles. Si la condition :

$$a_i^l + d^l(i, j) \geq b_j^l$$

est vérifiée pour une ressource  $l$ , alors l'arc  $(i, j)$  ne peut faire partie d'aucun chemin appartenant à la solution. Il peut alors être retiré du graphe initial. Il s'agit de la méthode présentée dans la section 2.5.2.

### 5.9.2 Nombre minimum de chemins

Comme proposé dans [Lar99], nous arrêtons prématurément l'exécution de l'algorithme quand des nombres suffisants de chemins et d'étiquettes totaux sont atteints. Cette réduction est particulièrement intéressante car quand aucun chemin n'est finalement trouvé, nous savons qu'il n'existe pas de chemin et nous n'avons pas à relancer d'autre algorithme. Il s'agit de la méthode présentée dans la section 2.5.2.

### 5.9.3 Réductions basées sur le coût réduit

Enfin, comme proposé dans [RGP02], nous réduisons également le graphe initial à partir de raisonnements sur le coût réduit. Il s'agit de la méthode présentée dans la section 2.5.3.

## 5.10 Benchmark de Solomon

Nous nous sommes intéressés aux instances de Solomon [Sol87] qui constituent une batterie de problèmes standard sur laquelle sont très souvent testées les méthodes exactes et heuristiques de résolution du VRPTW. Nous avons adopté les conventions utilisées par la plupart des méthodes exactes : minimisation de la distance uniquement sans se préoccuper du nombre de véhicules ; les distances et les temps de parcours sont déterminés par la distance euclidienne entre les paires de coordonnées  $(x, y)$ , arrondie au dixième inférieur. Il existe deux séries d'instances : la série 1 pour laquelle les fenêtres de temps sont étroites, la série 2 pour laquelle les fenêtres de temps sont plus larges. La série 1 est plus facile à résoudre parce que plus contrainte : les fenêtres de temps serrées limitent notamment le nombre de successeurs possibles pour une visite. C'est pourquoi la littérature s'est concentrée jusqu'à présent sur la série 1. Nous nous sommes particulièrement intéressés à la série 2 parce qu'elle contient encore de nombreuses instances ouvertes. Les instances de tests sont en outre divisées en trois classes : la classe "c" dans laquelle les visites sont divisées en plusieurs groupes compacts d'un point de vue géographique, la classe "r" dans laquelle les coordonnées des visites sont distribuées aléatoirement et la classe "rc" où une partie des visites sont groupées géographiquement et les autres sont placées aléatoirement. Chacune des instances est un problème à 100 visites duquel sont extraits deux problèmes réduits, l'un à 25 visites, l'autre à 50 visites.

Instances	nbVisits	$\frac{LB-LB(1)}{LB(1)}$	$\frac{LB(2)-LB(1)}{LB(1)}$	$\frac{LB-LB(1)}{OPT-LB(1)}$	$\frac{LB(2)-LB(1)}{OPT-LB(1)}$
R1	50	0,38%	0,33%	23,56%	36,86%
R1	100	0,32%	0,15%	13,95%	15,59%
RC1	50	0,55%	8,89%	11,62%	53,03%
RC1	100	0,55%	1,82%	3,06%	38,77%

TAB. 5.1 – Bornes inférieures sur la série 1

## 5.11 Résultats

Nous avons déjà présenté une partie de ces résultats dans [CDP02] et [Cha02c].

Quand il n'est plus possible de générer de nouvelle tournée de coût réduit favorable, la solution du problème maître avec contraintes d'intégralité relâchées nous donne une borne inférieure pour la branche en cours d'exploration.

Dans le cas du nœud racine, nous obtenons une borne inférieure globale. En utilisant un sous-problème qui ne génère que des tournées sans cycle, nous résolvons un problème plus contraint, et les bornes obtenues sont théoriquement égales ou meilleures. Le tableau 5.1 compare les bornes inférieures obtenues pour les séries R1 et RC1 avec celles présentées dans [CR99]. Les deux comparaisons données sont :

- entre notre borne inférieure  $LB$  (obtenue au nœud racine) et  $LB(1)$  de [CR99], qui correspond à un sous-problème n'éliminant que les cycles de type  $i - j - i$ .
- entre  $LB(2)$  et  $LB(1)$  de [CR99].  $LB(2)$  correspond à  $LB(1)$ , avec en plus, entre autres, des coupes d'élimination de sous-tours.

Nous voyons dans ce tableau que notre borne inférieure est effectivement meilleure que celle obtenue avec un sous-problème avec cycles. En pratique, l'écart entre notre borne et la solution optimale est de 17% inférieur à celui entre  $LB(1)$  et la solution optimale. Le même calcul pour  $LB(2)$  donnerait 47%, ce qui suggère qu'il pourrait être intéressant d'intégrer certaines des coupes de  $LB(2)$  à notre algorithme.

Aux autres nœuds les bornes inférieures locales obtenues nous permettent de ne pas explorer certaines branches, et donc de réduire le temps total.

Ces deux conséquences, entre autres, nous ont permis de fermer des instances auparavant ouvertes.

Le tableau 5.2 montre plus en détail l'amélioration des bornes inférieures obtenue sur des instances de la série 2. Les mêmes conventions sont utilisées que dans le tableau précédent, et les lignes en gras correspondent aux instances auparavant ouvertes que nous avons fermées.

Nous avons également essayé de représenter graphiquement cette différence de

Instance	nbVisits	LB	LB(1)	OPT	$\frac{LB-LB(1)}{LB(1)}$	$\frac{LB-LB(1)}{OPT-LB(1)}$
<b>R204</b>	<b>25</b>	<b>350,47</b>	<b>337,025</b>	<b>355,0</b>	<b>3,99%</b>	<b>74,80%</b>
<b>R208</b>	<b>25</b>	<b>328,20</b>	<b>321,752</b>	<b>328,2</b>	<b>2,00%</b>	<b>100,0%</b>
R201	50	791,90	788,625	791,9	0,42%	100,00%
R202	50	698,50	694,162	698,5	0,62%	100,00%
<b>R203</b>	<b>50</b>	<b>598,58</b>	<b>592,383</b>	<b>605,3</b>	<b>1,05%</b>	<b>48,00%</b>
<b>R205</b>	<b>50</b>	<b>682,85</b>	<b>668,524</b>	<b>690,1</b>	<b>2,14%</b>	<b>66,40%</b>
<b>R206</b>	<b>50</b>	<b>626,34</b>	<b>613,445</b>	<b>632,4</b>	<b>2,10%</b>	<b>68,05%</b>
<b>R209</b>	<b>50</b>	<b>599,83</b>	<b>585,939</b>	<b>600,6</b>	<b>2,37%</b>	<b>94,75%</b>
<b>R210</b>	<b>50</b>	<b>636,10</b>	<b>627,385</b>	<b>645,6</b>	<b>1,39%</b>	<b>47,85%</b>
R201	100	1140,30	1139,746	1143,2	0,05%	16,04%
<b>RC203</b>	<b>25</b>	<b>326,90</b>	<b>220,182</b>	<b>326,9</b>	<b>48,47%</b>	<b>100,0%</b>
<b>RC204</b>	<b>25</b>	<b>299,70</b>	<b>191,221</b>	<b>299,7</b>	<b>56,73%</b>	<b>100,0%</b>
<b>RC208</b>	<b>25</b>	<b>269,10</b>	<b>163,009</b>	<b>269,1</b>	<b>65,08%</b>	<b>100,0%</b>
RC201	50	684,80	678,867	684,8	0,87%	100,0%
<b>RC202</b>	<b>50</b>	<b>613,60</b>	<b>516,619</b>	<b>613,6</b>	<b>18,77%</b>	<b>100,0%</b>
<b>RC203</b>	<b>50</b>	<b>555,30</b>	<b>421,146</b>	<b>555,3</b>	<b>31,85%</b>	<b>100,0%</b>
<b>RC205</b>	<b>50</b>	<b>630,20</b>	<b>567,970</b>	<b>630,2</b>	<b>10,96%</b>	<b>100,0%</b>
<b>RC206</b>	<b>50</b>	<b>610,00</b>	<b>447,305</b>	<b>610,0</b>	<b>36,37%</b>	<b>100,0%</b>
<b>RC207</b>	<b>50</b>	<b>558,60</b>	<b>395,725</b>	<b>558,6</b>	<b>41,16%</b>	<b>100,0%</b>
RC201	100	1255,94	1244,456	1261,8	0,92%	66,22%
<b>RC202</b>	<b>100</b>	<b>1088,08</b>	<b>1012,616</b>	<b>1092,3</b>	<b>7,45%</b>	<b>94,70%</b>
<b>RC205</b>	<b>100</b>	<b>1147,61</b>	<b>1063,922</b>	<b>1154,0</b>	<b>7,87%</b>	<b>92,90%</b>

TAB. 5.2 – Bornes inférieures sur la série 2

bornes. Les figures 5.1 et 5.2 montrent les deux solutions relâchées pour l'instance *RC203* avec 25 véhicules. La figure 5.1 montre la solution correspondant à la borne *LB(1)* ne supprimant que les cycles de type  $i - j - i$ . Cette solution contient 16 routes, chacune de valeur inférieure à 1. La combinaison de routes utilisant des cycles de taille supérieure à 1 permet en effet d'obtenir des solutions relâchées de valeur totale inférieure à la solution sans cycle. La solution relâchée (en réalité entière et donc optimale) quand les cycles ne sont pas autorisés est illustrée dans la figure 5.2.

Le tableau 5.3 donne les résultats de notre schéma de coopération sur les 17 instances précédemment ouvertes dont nous avons prouvé l'optimum : la valeur de la solution optimale en terme de distance parcourue (colonne *OPT*), le nombre de véhicules pour cette solution, la différence relative entre la borne inférieure au nœud racine et la solution optimale (colonne *écart*), le nombre de nœuds explorés dans l'arbre du *branch-and-price*, le nombre de sous-problèmes résolus (colonne *SP*) et les temps de résolution sur un bi-processeur (2x800 MHz, mais un seul processeur est réellement utilisé). Deux temps sont donnés : celui pour trouver la solution optimale et celui pour réaliser la preuve d'optimalité. Les instances marquées par (\*) et (\*\*) sont celles pour lesquelles nous trouvons une solution optimale meilleure que celles données respectivement par [KLM01] et par [Lar99] et [KLM01]. Sur ces instances, nos solutions ont été validées par Jesper Larsen.

Les temps indiqués dans cette section sont obtenus sur un Pentium IV cadencé à 1,5 Ghz avec 256 Mo de mémoire et avec un prototype écrit en Java de notre environnement de génération de colonnes et de coupes utilisant ILOG JNI CPLEX 7.5, ILOG JSOLVER 1.0 et la JVM d'IBM pour Linux version 1.3.0.

## Conclusions

Nous avons étudié dans cette section comment les méthodes de génération de colonnes avec sous-problème de plus court chemin peuvent s'appliquer aux problèmes de tournées de véhicules. Nous avons également présenté une contribution originale consistant à utiliser un algorithme de plus court chemin élémentaire quand cela correspondait au modèle d'origine. Alors que l'ensemble des publications avait jusqu'alors rejeté cette possibilité, l'algorithme correspondant étant trop inefficace, nous lui avons appliqué diverses améliorations qui permettent de rendre son utilisation efficace.

La modification de l'algorithme de plus court chemin pour correspondre à la définition exacte du sous-problème n'est pas toujours possible. De plus, contrairement au cas du VRP, il ne sera pas toujours possible de relaxer les contraintes incompatibles avec l'algorithme de plus court-chemin à base d'étiquettes. Il sera alors nécessaire de mettre en œuvre d'autres méthodes pour incorporer ces va-

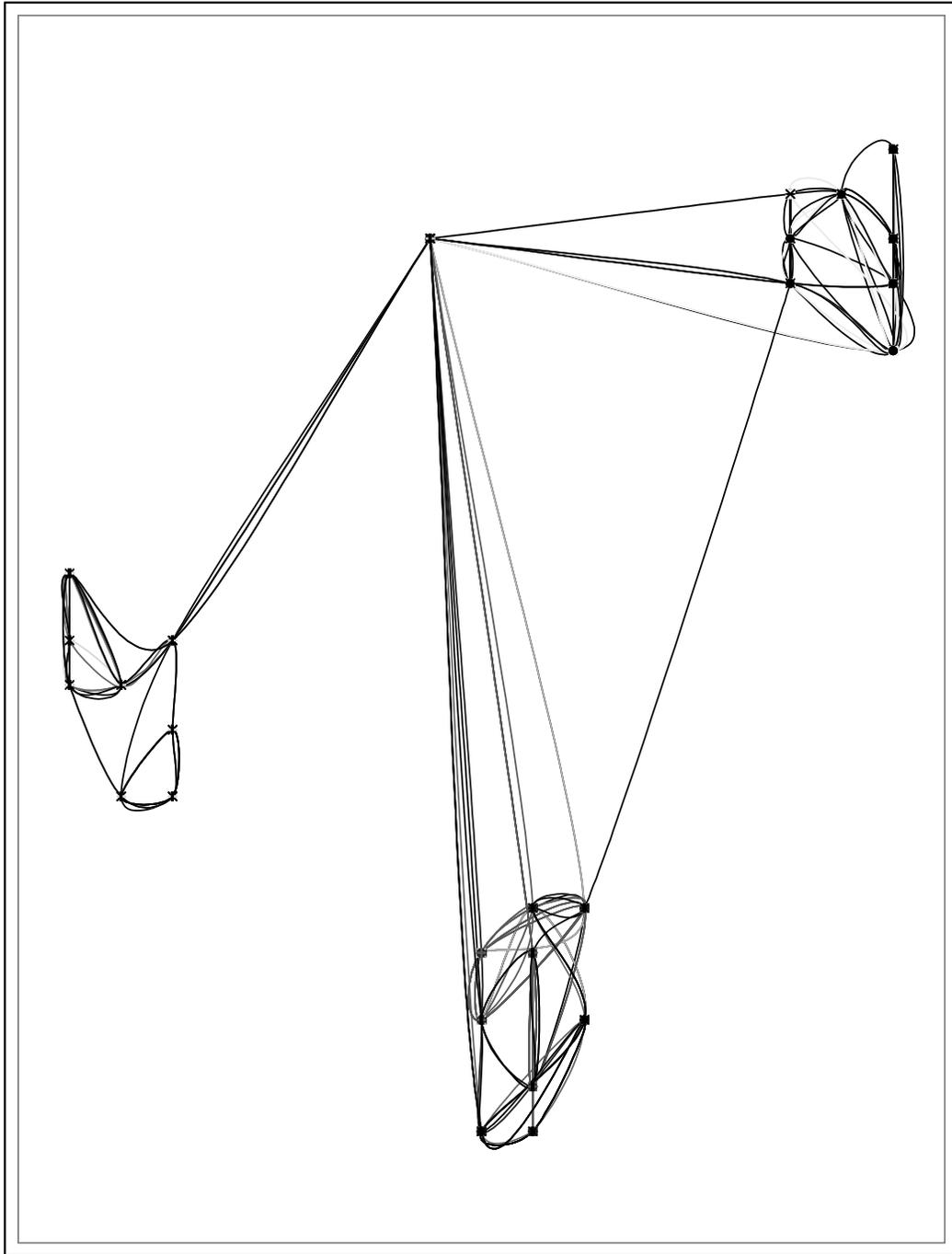


FIG. 5.1 – LB(1) pour RC203.25

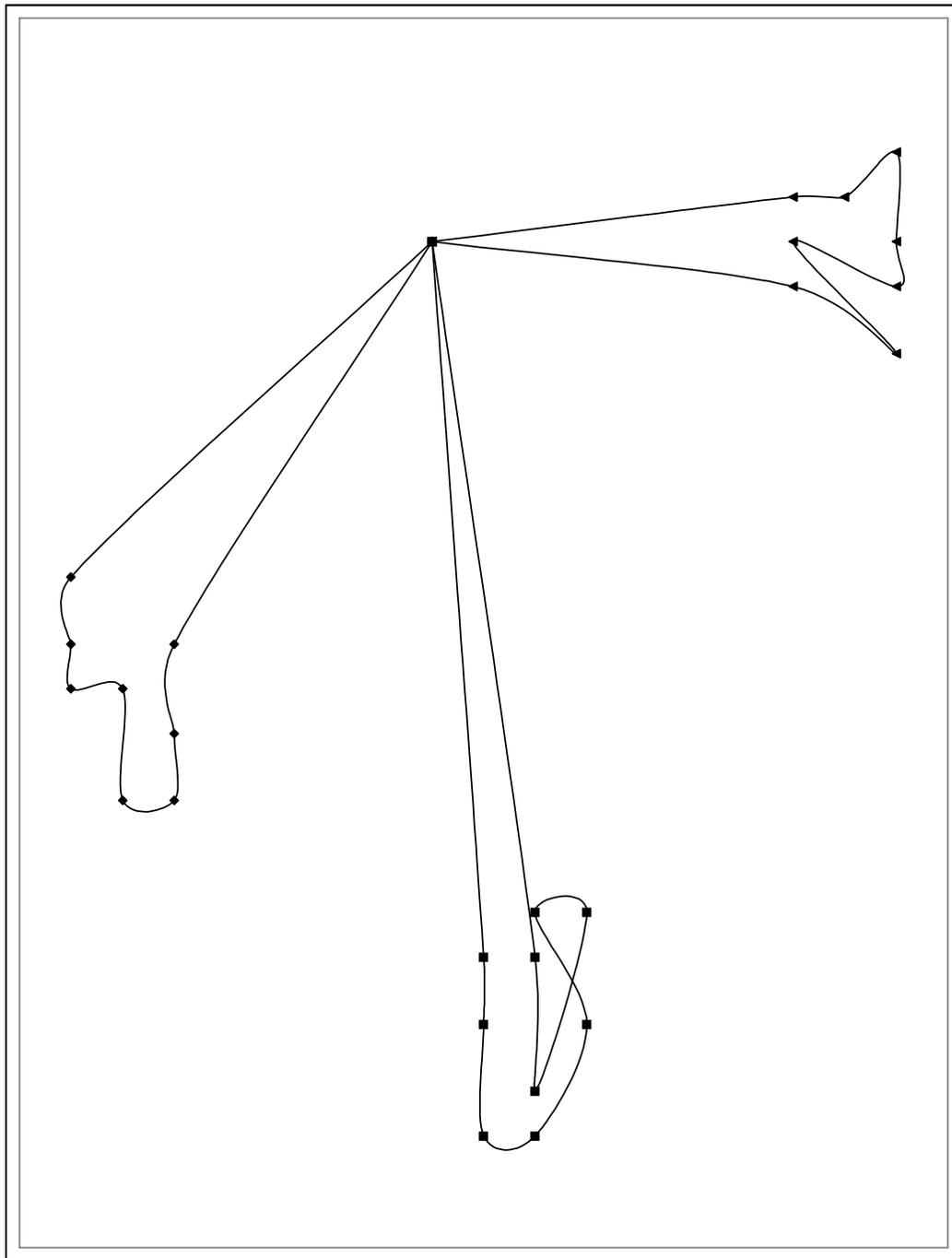


FIG. 5.2 – LB pour RC203.25

Instance	OPT	V.	écart (%)	N.	SP	$T_{opt}$	$T_{preuve}$
<i>R203.50</i>	605,3	5	1,11	15	432	117,8	3320,9
<i>R204.25</i>	355,0	2	1,27	17	365	88,4	171,6
<i>R205.50**</i>	690,1	4	1,05	95	1551	301,4	531,0
<i>R206.50</i>	632,4	4	0,96	67	1319	2972,8	4656,1
<i>R208.25*</i>	328,2	1	0,00	1	64	705,7	741,5
<i>R209.50</i>	600,6	4	0,13	5	144	120,8	195,4
<i>R210.50</i>	645,6	4	1,47	977	16183	25151,1	65638,6
<i>RC202.50</i>	613,6	5	0,00	1	70	5,3	13,0
<i>RC202.100</i>	1092,3	8	0,39	39	1861	18053,8	19636,5
<i>RC203.25*</i>	326,9	3	0,00	1	65	4,0	5,1
<i>RC203.50</i>	555,3	4	0,00	1	315	4479,4	4481,5
<i>RC204.25</i>	299,7	3	0,00	1	58	1,95	13,0
<i>RC205.50*</i>	630,2	5	0,00	1	82	10,3	10,6
<i>RC205.100</i>	1154,0	7	0,55	71	2706	3131,6	15151,7
<i>RC206.50</i>	610,0	5	0,00	1	61	8,6	9,4
<i>RC207.50</i>	558,6	4	0,00	1	107	66,0	71,1
<i>RC208.25</i>	269,1	2	0,00	1	185	32239,3	33785,3

TAB. 5.3 – Les instances que nous avons fermées

riations. Dans le prochain chapitre, nous étudions l'application des méthodes de génération de colonnes avec sous-problème de plus court chemin aux problèmes de planification de ressources et en particulier au problème de génération de *pairings*. Nous verrons que dans de nombreux cas interviennent des contraintes additionnelles plus complexes et nous proposerons diverses autres méthodes pour les prendre en compte.

# Chapitre 6

## Résolutions heuristiques des sous-problèmes : application à la planification de ressources

Dans le chapitre précédent, nous avons proposé une modification du modèle de plus court chemin avec contraintes de ressources et fenêtres de temps couramment utilisé en génération de colonnes, ainsi que de l'algorithme à base d'étiquettes utilisé pour le résoudre, afin de prendre en compte la contrainte d'élémentarité des chemins. Cette modification a été appliquée à un problème de tournées de véhicules. Certains sous-problèmes peuvent cependant comporter d'autres types de contraintes qui ne sont pas directement compatibles avec cet algorithme, i.e. qui ne prennent pas la forme d'une réduction du graphe ou d'une fenêtre de temps. Ceci est souvent le cas dans les problèmes de planification de ressources soumis aux compagnies aériennes.

Dans ce chapitre, nous présentons l'application des méthodes de génération de colonnes avec sous-problème de plus court chemin à un problème de planification de ressources apparaissant dans l'industrie du transport aérien : le problème de *génération de pairings*. Cette application des méthodes de génération de colonnes est, dans sa version la plus simplifiée, parmi les plus étudiées. Dans sa version plus réaliste, le sous-problème inclut des contraintes additionnelles qui nous ont amené à mettre en œuvre des techniques de programmation par contraintes, comme cela a déjà été proposé pour d'autres situations similaires. Cependant, nous proposons également plusieurs contributions originales :

- utilisation d'un niveau de filtrage supérieur utilisant l'algorithme de plus court chemin avec contraintes de ressources et fenêtres de temps à base d'étiquettes,
- modification de l'algorithme à base d'étiquettes pour prendre en compte d'autres schémas de coûts,

- utilisation d’une *heuristique d’expert* pour guider la recherche,
- utilisation de stratégies de recherche pour diversifier la partie de l’arbre de recherche visitée par l’heuristique précédente,

Ces améliorations profitent à la programmation par contraintes quand celle-ci est utilisée dans le cadre de la génération de colonnes. La contrainte globale permet en effet, d’envisager une utilisation efficace de la programmation par contraintes même si le sous-problème de plus court chemin est pur (i.e. ne contient pas de contraintes additionnelles incompatibles). D’autre part, l’utilisation combinée d’heuristiques et de stratégies de recherche, permet de diversifier grandement les types de colonnes ajoutées et ainsi de stabiliser la génération de colonnes.

Ce chapitre, qui reprend en partie des idées et résultats présentés dans [Cha99a] et [Cha99c], est organisé de la façon suivante. La première section donne une vision d’ensemble des problématiques existant dans l’industrie du transport aérien. Un modèle de génération de colonnes est donné pour le problème particulier de la génération de pairings. Les trois sections suivantes proposent des techniques pour prendre en compte les spécificités de ce problème. Les sections 6.2 et 6.3 présentent respectivement deux modifications de l’algorithme à base d’étiquettes pour prendre en compte des schémas de coût différents et la contrainte sur les *bases*. La section 6.4 propose plusieurs idées pour traiter les autres contraintes additionnelles ne pouvant être intégrées directement à l’algorithme à base d’étiquettes. Enfin, la section 6.5 présente des résultats sur des applications réelles de génération de pairings.

Les résultats présentés dans ce chapitre ont été en partie obtenus lors du développement d’une application réelle réalisée avec Javier Lafuente.

## 6.1 Problématique des compagnies aériennes

Nous allons illustrer nos propositions sur un des problèmes réels auxquels sont confrontées les compagnies aériennes. Dans cette section, nous donnons tout d’abord une vision d’ensemble des problèmes existant dans cette industrie, puis nous présentons un problème particulier, le problème de génération de *pairings*. Enfin, nous donnons un modèle de génération de colonnes qui fera apparaître des sous-problèmes de plus court chemin comportant des contraintes et formats de coûts plus complexes que ceux du chapitre précédent.

### 6.1.1 Les problèmes d’optimisation dans les compagnies aériennes

Chaque compagnie aérienne est amenée à résoudre une série de problèmes complexes de planification de ses avions, de ses pilotes et de ses membres d’équipage

(sans compter les activités au sol, qui sont généralement traitées à part et que nous n'aborderons pas). Ce type de planification a été historiquement un de ceux sur lesquels le plus grand nombre d'études dans le domaine de la recherche opérationnelle et de l'optimisation ont été réalisées. De nombreuses compagnies aériennes disposent même de leur propre département de recherche opérationnelle, responsable de fournir aux autres départements des outils de planification efficaces. Ce profond intérêt est simplement dû à la possibilité de réaliser d'importantes économies. En effet, l'économie d'un seul avion pour effectuer un même ensemble de vols peut supposer d'énormes réductions de coûts opérationnels pour la compagnie.

Au delà des problèmes de planification, citons également que les compagnies aériennes sont également amenées à utiliser, auparavant, des outils de simulation, et finalement, durant l'exécution de la planification, des outils de réaction aux événements et de re-planification facilitant la mise en œuvre des corrections dues aux très fréquents changements de dernière minute. En effet, il ne faut pas oublier qu'une planification, aussi optimale soit-elle, peut être globalement remise en question pour de simples raisons météorologiques.

La planification des activités aériennes d'une compagnie se divise généralement en une suite de processus résolus séquentiellement :

- choix des flottes et des vols en fonction des estimations des demandes de voyages et des choix stratégiques de la compagnie,
- planification des séquences de vols pour les avions. Cette phase, ainsi que les suivantes sont réalisées indépendamment pour chacune des flottes d'avions (par exemple, Boeing 737 et Airbus A320).
- création de *pairings*, séquences de vols sur plusieurs jours anonymes (i.e. n'étant pas attribuées nominativement à des équipages) et ne tenant pas compte des activités particulières, comme les repos, les visites médicales, les entraînements, etc. Cette phase est connue sous le nom de *génération de pairings*. Il s'agit du problème que nous allons traiter.
- affectation des *pairings* à des membres d'équipages complets. Cette phase est connue sous le nom de *Crew Rostering*.

Il est compréhensible que si l'ensemble du problème pouvait être traité globalement, des solutions de meilleure qualité pourraient être trouvées. Cependant, après chaque phase, les décisions prises dans les précédentes ne sont jamais remises en question. Cette limitation peut paraître importante, mais elle est nécessaire au vu de la taille des données des problèmes traités. Quelques recherches ont cependant été dédiées à des tentatives pour grouper certains de ces processus. De plus amples descriptions des problèmes existants dans l'industrie du transport aérien sont données dans [AFP98], [VBJN95], [VBJ<sup>+</sup>96], [Erd99] et [Yu98].

### 6.1.2 Génération de pairings

Le problème traité ici correspond à la troisième phase qui consiste à générer un ensemble de *pairings* couvrant un plan de vol et respectant un ensemble de réglementations aériennes et de conventions collectives propres à la compagnie.

Trois niveaux d'éléments à planifier sont manipulés :

- les *vols*, simples données provenant de phases antérieures du processus. A un vol sont associés des aéroports de départ et d'arrivée, des horaires de départ et d'arrivée.
- les *activités aériennes*, séquences de vols ne permettant aucun repos intermédiaire, ne s'étendant en général que sur une durée inférieure à la journée,
- les *pairings*, séquences d'*activités aériennes* et de repos, ayant pour aéroport d'origine et de fin un même aéroport appelé *base*. Ce *pairing* ne pourra être affecté qu'à un membre d'équipage associé à cette base.

La figure 6.1 montre deux séries de vols pour deux avions *AVION* – 1 et *AVION* – 2. Un *pairing* est représenté, comme succession de deux activités aériennes. La première activité aérienne contient les vols *VOL* – 1 et *VOL* – 2 et la deuxième *VOL* – 11 et *VOL* – 12.

#### Exemples de réglementations

Nous illustrons ici les diverses réglementations et conventions possibles en donnant quelques exemples de règles intervenant dans le problème que nous avons traité. Les réglementations sont qualitativement identiques d'une compagnie à une autre pour une même zone géographique (Europe ou États-Unis). Les règles que nous présentons ici correspondent à une compagnie de type européenne. Parmi les principales réglementations, citons :

- une table définit la durée maximale d'une activité aérienne en fonction du nombre de vols et de l'heure locale de décollage du premier vol de l'activité,
- le temps de repos entre deux activités aériennes doit être supérieur à la durée de la précédente activité aérienne et à une durée minimale de repos égale à 10 heures 15 minutes.

D'autre part, il existe également des conventions propres à la compagnie aérienne considérée. Certaines compagnies négocient financièrement avec des syndicats de pilotes la non-application de certaines conventions. Les méthodes d'optimisation prennent ici plus d'importance, car elles permettent, en effectuant plusieurs planifications, d'estimer le coût de la prise en compte ou non d'une convention. Parmi ces conventions citons, dans le cas que nous avons traité :

- le changement d'avion (l'exécution de deux vols successifs correspondants à deux avions différents), n'est permis à l'intérieur d'une même activité aérienne que si ces deux vols sont séparés d'un minimum de 75 minutes,

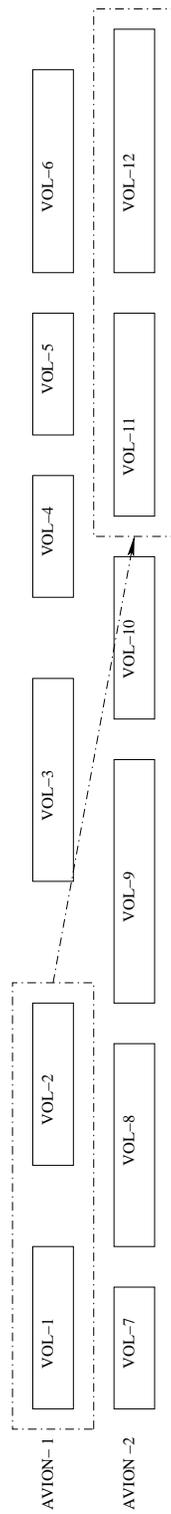


FIG. 6.1 – Exemple de vol, activité aérienne et pairing

- le changement d’aéroport, entre deux activités aériennes est envisageable s’il existe un vol, dit *vol de situation*, permettant au membre d’équipage de se déplacer, comme simple passager, d’un aéroport à l’autre. Ce temps de déplacement doit être cependant pris en compte dans le calcul de la durée de l’activité aérienne suivante, et donc du temps de repos lui succédant.
- les activités aériennes ne peuvent commencer et/ou finir que dans certains aéroports particuliers appelés *aéroports de situation*. En effet, la compagnie doit disposer dans ces aéroports de structures permettant la prise en charge et le repos des membres d’équipage. Cet ensemble, en général restreint, est plus grand que l’ensemble des bases. Cette convention n’est dans la pratique pas très restrictive. En effet, des activités terminant dans des aéroports qui ne sont pas de situation, ne correspondent en général à aucune autre activité ultérieure suffisamment proche permettant de prolonger celle-ci en un pairing valide.

### Aperçu des méthodes utilisées

Plusieurs niveaux de décisions existent, sur lesquels il est difficile d’effectuer simultanément une recherche : les activités aériennes et les pairings. L’idéal serait de chercher une solution parmi tous les pairings réalisables utilisant toutes les activités aériennes réalisables. Ici aussi, le problème est habituellement décomposé. Cependant, cette décomposition est parfois réalisée sans pertes, lorsque toutes les activités aériennes peuvent être générées et utilisées dans la génération des pairings.

Le processus consiste alors à :

- une phase de génération de l’ensemble des activités aériennes possibles, celles-ci pouvant se recouvrir entre elles. Ceci est un processus séquentiel, complet.
- une phase d’optimisation proprement dite, qui consiste à générer des pairings (séquences d’activités aériennes provenant de la phase précédente), couvrant les vols et ayant un coût minimum,

Par exemple, deux activités couvrant le même vol peuvent être générées dans la première phase, mais ne peuvent faire partie de deux pairings sélectionnés dans la deuxième phase.

### Génération des activités aériennes

Cette procédure ne comporte aucune phase d’optimisation. Elle consiste simplement à générer toutes les activités aériennes possibles, et à ne conserver que celles vérifiant les contraintes propres à ce niveau. C’est une procédure rapide dont la seule difficulté réside dans l’augmentation exponentielle du temps d’exécution en fonction du nombre de vols. Dans notre cas, la table 6.4 montre que le nombre

d'activités aériennes n'est pas beaucoup plus grand que le nombre de vols. Ceci est du en particulier à la restriction sur les aéroports dans lesquels peuvent commencer ou finir des activités aériennes et à la proportion entre le nombre de ces aéroports et le nombre total d'aéroports visités.

### Génération des pairings optimaux

C'est dans cette phase qu'a lieu la véritable optimisation sous forme de procédure de génération de colonnes. L'ensemble des activités aériennes étant donné, il s'agit de trouver un ensemble de pairings de coût minimal tel que l'ensemble des activités aériennes réalisées recouvre l'ensemble des vols.

#### 6.1.3 Modèles de génération de colonnes

Nous allons maintenant établir un modèle de génération de colonnes pour le problème de génération de pairings. Nous établirons en fait une série de modèles : un modèle de base auquel seront associés plusieurs schémas de coûts possibles, et un ensemble de contraintes additionnelles pouvant lui être ajoutées.

#### Discussions préliminaires

Les coûts utilisés sont toujours propres au pairing, i.e. le coût total est toujours la somme de coûts individuels associés à chaque pairing. Cependant, ce coût individuel n'a pas toujours la forme d'une accumulation simple d'une ressource comme cela était le cas pour le VRP. En effet, un *petit* pairing ne comportant que peu de travail à réaliser (peu d'heures de vols par exemple) peut cependant correspondre à un coût important pour la compagnie. Le schéma de coût alternatif consiste alors à utiliser le maximum de différents coûts simples de type accumulation d'une ressource. Par exemple, le coût peut être défini comme le maximum entre :

- un coût fixe de 40,
- 10 fois le nombre de jours couverts par le pairing,
- le nombre d'heures de vols du pairing.

D'autre part, nous considérerons quelques contraintes supplémentaires pouvant être ajoutées au modèle de base. Parmi celles-ci, citons la contrainte sur les bases, obligeant les pairings à commencer et finir dans un même aéroport pris parmi l'ensemble des bases. Cette contrainte sur les bases est présente dans la majorité des problèmes de génération de pairings.

Une autre contrainte additionnelle, moins souvent traitée, est la prise en compte de vols de situation dans les temps de vols utilisés pour définir les temps de repos minimaux. En effet, la réalisation d'une activité aérienne  $k$  après une

activité aérienne  $i$  ayant des aéroports discordants peut rendre nécessaire l'utilisation d'un vol de situation. Dans le cas où ce vol de situation est effectué juste avant l'activité aérienne  $k$ , il doit être pris en compte en partie dans le temps total de l'activité à utiliser pour calculer le repos devant lui succéder. Cette succession peut alors rendre impossible qu'une troisième activité  $l$  soit réalisée après  $k$ . Ceci est illustré dans les figures 6.2 et 6.3. La première montre le cas où aucun vol de situation n'est nécessaire, le temps de repos après  $k$  permet alors de réaliser l'activité  $l$ . Au contraire, pour que  $k$  suive  $i$ , il est nécessaire de passer par un vol de situation qui augmente le temps de repos minimum nécessaire et rend impossible la réalisation de l'activité  $l$  après l'activité  $k$ .

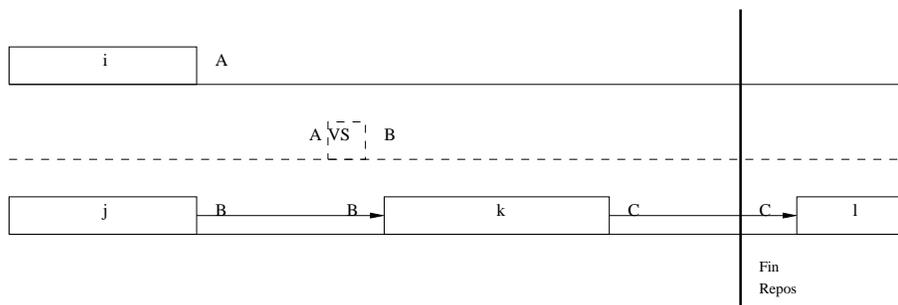


FIG. 6.2 – Le vol de situation n'est pas nécessaire.

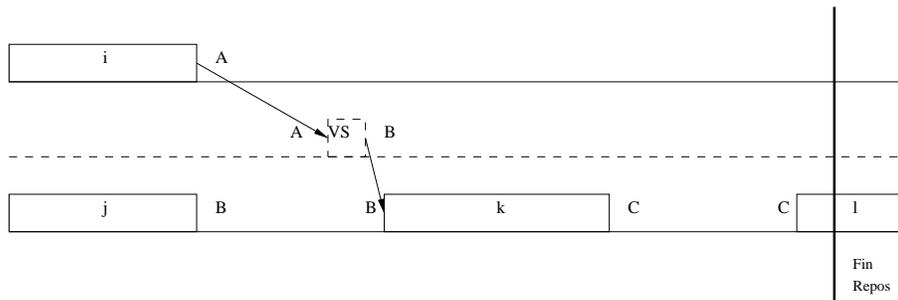


FIG. 6.3 – Le vol de situation est nécessaire.

Ces deux types de contraintes nous intéressent particulièrement car elles rompent la structure de l'algorithme de plus court chemin à base d'étiquettes.

### Modèle du sous-problème

Le sous-problème est ici aussi un problème de plus court chemin dans un graphe défini par les activités. Nous utilisons donc des variables *next* comme pour

le VRP. Ici, les domaines des variables sont plus restreints, l'ensemble des activités suivant une activité donnée étant plus limité. Par exemple, le domaine de  $next_i$  ne contient pas les activités antérieures à  $i$  ou bien se recouvrant temporellement avec  $i$ . Le problème de l'existence de cycles dans les chemins ne se posera donc pas, le graphe utilisé étant acyclique. Les domaines sont également réduits en fonction des aéroports de départ et d'arrivée. Une activité terminant dans l'aéroport A ne peut bien sûr n'être suivie que d'une activité commençant dans l'aéroport A ou possédant un vol de situation convenable.

Les ressources correspondent au nombre d'heures d'activité, et au nombre de jours calendaires. Seul le nombre de jours calendaires est limité, mais les deux ressources peuvent être utilisées dans les coûts.

$$\begin{aligned}
 SP &= (X, D, C, O(\text{minimise}, cr(D))) \\
 X &= (D \cup cumul_{ha} \cup cumul_j) \\
 D &= (next_0, next_1, \dots, next_{N+1}) \\
 C &= \left( \begin{array}{l} pathlength(next, cumul_{ha}, dist_{ha}) \\ pathlength(next, cumul_j, dist_j) \\ cumul_{j,N+1} \leq M_j \end{array} \right)
 \end{aligned}$$

Nous avons vu que plusieurs types de schémas de coûts différents sont utilisés, prenant la forme d'un maximum entre plusieurs termes dont chacun est un terme linéaire d'une accumulation de ressource. Quand un seul terme est pris en compte, nous sommes réduits à un coût du même type que dans le VRP. La forme générale du coût est :

$$sc(D) = \max_l(\alpha_l \cdot cumul_{l,N+1})$$

Les contraintes additionnelles sur les temps de repos à prendre en compte lors de l'utilisation de vols de situation sont définies après utilisation d'un pré-traitement. Avant la recherche, tous les triplets  $(i, k, l)$  correspondant à l'exemple donné peuvent être obtenus de manière systématique. Nous avons alors pour chacun de ces triplets une contrainte interdisant la transition de la forme :

$$(next_i = k) \Rightarrow (next_k \neq l)$$

Ces contraintes ne sont bien sûr pas compatibles avec l'algorithme à base d'étiquettes. Dans ce chapitre, nous proposons d'utiliser la programmation par contraintes afin de résoudre des problèmes comportant ce type de contraintes.

### Modèle du problème maître

Nous notons  $V$  l'ensemble des vols à couvrir,  $A$  l'ensemble des activités aériennes, et pour tout  $i \in V$ ,  $A_i$  le sous-ensemble des activités aériennes contenant le vol  $i$ .

$$\begin{aligned}
PGC &= (PM, SP) \\
PM &= (OM, CM_i) \\
OM &= (\text{minimise}, sc(D)) \\
CM_i &= ((1, 1), \sum_{a \in A_i} \text{next}_a \neq a), \forall i \in V
\end{aligned}$$

### 6.1.4 Autres problèmes de planification de ressources

Ce problème de génération de pairings n'est qu'un cas particulier de problème de planification de ressources. Nous incluons dans cette catégorie l'ensemble des problèmes consistant à affecter certaines activités dont les dates et conditions d'exécution ont été fixées auparavant à différents types de ressources dont les règles d'utilisation sont également fixées. Par règle d'utilisation nous entendons l'ensemble des contraintes s'appliquant sur les séquences d'activités qu'une même ressource peut réaliser. Dans notre exemple de génération de pairings, il s'agit de l'ensemble des règlements et conventions définissant la validité d'un pairing.

Les problèmes de planification sont très nombreux dans la vie courante. Uniquement dans le cas des compagnies aériennes, de nombreux autres problèmes apparaissent. La planification des ressources au sol est également importante : personnel de maintenance des avions juste avant et après les vols, de nettoyage de l'intérieur, de déplacement des bagages, de facturation des passagers. Toutes ces ressources doivent couvrir un ensemble de tâches définies à l'avance. En effet, la planification des vols étant donnée, certaines règles permettent d'obtenir un ensemble d'activités lui correspondant.

Un autre domaine où des méthodes d'optimisation sont de plus en plus fréquemment utilisées pour résoudre des problèmes de planification de ressources est celui des *centres d'appel*. De nombreuses entreprises offrant des services au grand public ont de plus en plus tendance à limiter leur contacts avec leur client via des *numéros verts*. Derrière ces numéros de téléphone, des *centres d'appel* regroupent de nombreux opérateurs. En fonction d'une étude statistique, une courbe de demande est définie devant être couverte par des opérateurs dont les conditions de travail sont réglementées.

## 6.2 Utilisation de schémas de coûts différents

Une première spécificité à traiter est l'existence de schéma de coût de type :

$$\max_l(\alpha_l \cdot \text{cumul}_{l, N+1})$$

Nous montrons dans cette section que l'algorithme de plus court chemin avec étiquettes peut être adapté pour prendre en compte ce format de coût.

### 6.2.1 Format de coût utilisé en *Crew Scheduling*

Les réglementations et conventions prises en compte pour la validation des pairings sont très restrictives. Il est souvent nécessaire d'avoir recours à une variété importante de séquences d'activités aériennes : plus ou moins longues, contenant plus ou moins d'heures de vol, etc. D'autre part, toutes les activités ne sont pas équivalentes : vols de nuits, repos longs ou juste réglementaires, nuits passées hors bases, etc. Le format utilisé pour attribuer un coût à un pairing est alors plus complexe que celui utilisé par exemple dans les problèmes de tournées de véhicules. Par exemple, au delà du coût correspondant à l'occupation d'une chambre d'hôtel pour une nuit passée hors de la base, cet inconvénient peut parfois être compensé financièrement au membre d'équipage. De même, des pilotes occupés toute une journée pour très peu d'heures de vol peuvent cependant être payés de manière forfaitaire pour la journée.

La coutume est alors de définir un ensemble de coûts prenant en compte ces différents aspects. Le coût final associé au pairing est alors le maximum de ces coûts. Ainsi, même si le nombre d'heures est faible, le coût constant concerté permet d'offrir une compensation au membre d'équipage.

Chacun des coûts pris en compte dans ce calcul a cependant la forme d'un terme linéaire d'une accumulation de ressources.

### 6.2.2 Coût-réduit

Une propriété intéressante de cette forme de coût est qu'elle est conservée par le coût réduit. En effet, le coût réduit correspondant peut être simplifié et prendre la même forme :

$$\begin{aligned} cr(D) &= \max_l(\alpha_l \cdot \text{cumul}_{l,N+1}) - \text{dual}(D) \\ &= \max_l(\alpha_l \cdot \text{cumul}_{l,N+1} - \text{dual}(D)) \end{aligned}$$

### 6.2.3 Modification de l'algorithme

Il est facile de voir que la règle de dominance utilisée dans l'algorithme avec étiquettes est compatible avec ce format de coût réduit. En effet, si nous imaginons deux chemins partiels  $c_1$  et  $c_2$  arrivant au même nœud  $i$  et si  $c_1$  domine  $c_2$  (sur l'ensemble des ressources et sur le coût), alors  $c_2$  peut être éliminé. En effet, si  $c_2$  possédait un prolongement intéressant  $c_2^*$ , il serait possible de prolonger  $c_1$  de la même manière en  $c_1^*$  et voir que le coût réduit de cette prolongation serait meilleur.

Le format de coût précédemment défini peut donc être utilisé pour l'algorithme avec étiquettes. C'est ce que nous avons fait dans notre implantation. Dans le cas général, il faudrait conserver chacun des termes de  $max_i$ . Ici, les termes sont des accumulations de ressources et sont donc déjà conservés.

### 6.3 Traitement des bases

Une deuxième modification possible du problème correspond à la prise en compte des bases. Celle ci peut être traitée de deux manières différentes.

Tout d'abord, une méthode similaire à celle utilisée pour les différents dépôts dans le cas des problèmes de tournées de véhicules peut être utilisée. Un sous-problème différent est alors défini pour chacune des bases. Pour chaque sous-problème, les arcs sortants du nœud correspondant au début de la séquence ne seront conservés que si le premier vol possède comme aéroport de départ la base considérée. La même modification est réalisée pour le nœud d'arrivée.

Cependant, dans le cas présent, les distanciers utilisés sont identiques pour les différentes bases. La même étiquette peut être utilisée pour deux bases différentes si des vols de situation sont disponibles pour commencer la séquence de vols depuis chacune des bases. L'augmentation du nombre d'étiquettes peut alors être limitée en prenant en compte les différentes bases simultanément dans un même problème résolu avec un algorithme utilisant des étiquettes. Une notion d'*état* est alors ajoutée à l'algorithme à base d'étiquettes qui permet d'associer une unique étiquette à plusieurs chemins partiels identiques mais correspondant à des bases différentes. A chaque étiquette est associé un état représentant l'ensemble des bases possibles. Cet état est donc défini comme :

$$S = \{b_i\}$$

et une étiquette est maintenant définie par :

$$E_c = (C_c, D_c^1, \dots, D_c^L, S_c)$$

Les états sont utilisés pour limiter les transitions possibles entre nœuds. Une étiquette  $E_c$  finissant au nœud  $i$  peut être prolongée en une étiquette  $E_{c^*}$  au nœud  $j$ , si et seulement si :

$$S_{c^*} \subseteq S_c \text{ et } \forall b \in S_{c^*}, (i, j) \in X_b$$

avec  $X_b \subseteq X$  ensemble des arcs autorisés pour la base  $b^1$ .

---

<sup>1</sup>Cette méthode peut être étendue au cas où  $S_{c^*} \not\subseteq S_c$  pour prendre en compte des transitions d'états.

Un critère d'élimination des étiquettes peut ensuite être défini. Si  $E_1$  et  $E_2$  sont deux étiquettes correspondant à des chemins partiels finissant au même nœud  $i$ , et que  $E_1$  domine  $E_2$ , alors l'étiquette  $E_2$  peut être éliminée si et seulement si  $S_2 \subseteq S_1$ . Dans le cas contraire, si  $S_2 \not\subseteq S_1$ , l'état de  $E_2$  peut cependant être réduit à  $S_2 \setminus S_1$ .

L'utilisation de ces états permet de réduire le nombre total d'étiquettes créées par rapport à l'utilisation de plusieurs modèles indépendants. Une étiquette au nœud final peut même posséder un état correspondant à plusieurs bases. Celle-ci permet alors de créer plusieurs colonnes, une pour chaque base.

Nous n'avons pas utilisé cette technique des états dans notre implantation. En effet, nous devons prendre en compte une contrainte maître limitant le nombre total de pairings pour chaque base. Le schéma de coût réduit est alors différent pour chaque base, celui-ci devant prendre en compte la valeur duale de cette contrainte maître. Les étiquettes de même chemin partiel et de bases différentes n'ont alors plus les mêmes coûts réduits et ne peuvent donc plus être groupées.

## 6.4 Contraintes additionnelles non compatibles

Nous présentons ici des contraintes non nécessairement compatibles avec l'algorithme de plus court chemin avec fenêtres de temps. Contrairement au cas précédent des bases, nous ne connaissons pas de techniques permettant de traiter efficacement ces contraintes en conservant un algorithme à base d'étiquettes. Nous proposerons alors trois techniques pour prendre en compte ces contraintes dans la résolution du sous-problème :

- programmation par contraintes,
- utilisation d'heuristiques d'expert,
- utilisation de stratégies de recherche.

### 6.4.1 Contraintes non compatibles

#### Contraintes conditionnelles

Dans l'application la plus naturelle du problème de plus court chemin : le trajet le plus court pour rejoindre deux positions dans un réseau routier, des contraintes comme les sens interdits et les changements de direction interdits doivent être prises en compte. Le premier cas est le plus facile à traiter : le sous-problème étant simplement modifié en éliminant l'arc interdit, le problème conserve sa structure permettant d'utiliser un algorithme simple. Dans le deuxième cas, la contrainte s'écrit :

si  $(next[i]==j)$  alors  $(next[j]!=k)$

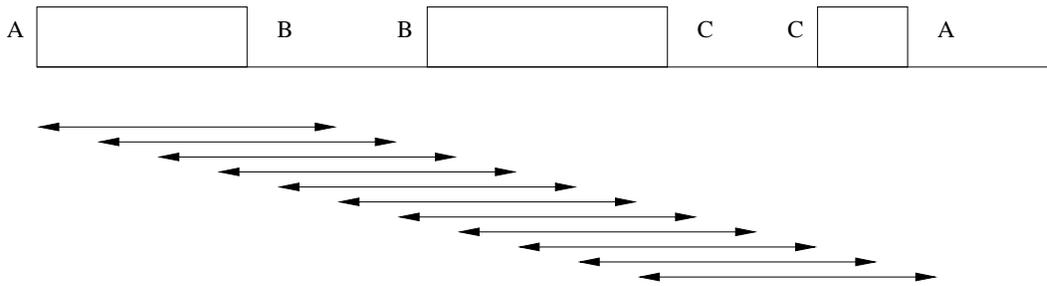


FIG. 6.4 – Limitation s'appliquant sur chaque intervalle

Nous avons vu que ce type de contrainte apparaît dans notre problème de génération de pairings lorsque les durées des vols de situation sont prises en compte pour calculer le temps de repos minimum nécessaires.

### Contraintes glissantes

Au delà de la simple limitation s'appliquant à une ressource précise en un nœud précis du graphe, il est fréquent d'avoir des contraintes limitant l'accumulation d'une ressource sur un ensemble d'ensembles de nœuds. Par exemple, une contrainte fréquente dans le domaine du transport aérien a la forme : *l'accumulation d'heures de vols sur  $N$  jours est limitée à  $H$  heures*. De même un minimum peut être imposé : *au moins  $M$  minutes de pause doivent être accumulées sur une frange de  $H$  heures de travail*. La figure 6.4 montre bien que même pour un petit pairing, la contrainte s'applique sur un intervalle de temps glissant. Pour chaque intervalle représenté, la limitation s'applique.

Chacune de ces contraintes pourrait utiliser la modélisation plus traditionnelle des ressources, mais nécessiterait autant de ressources que d'intervalles.

### 6.4.2 Sous-problème avec PPC

La partie du sous-problème correspondant à ce type de contraintes non compatibles est en général faible. Une grosse sous-partie du problème correspond au problème de plus court chemin avec fenêtres de temps et contraintes de ressources. Il serait alors dommage de ne pas profiter de l'existence d'un algorithme efficace pour résoudre cette partie du problème. Une méthode récemment proposée pour traiter ce type de situation consiste à intégrer l'algorithme traitant efficacement une certaine sous-partie du problème dans une contrainte globale dans un environnement de programmation par contraintes.

### Contrainte globale en programmation par contraintes

Cette méthode a déjà été appliquée dans le cas où la partie du problème correspondant à un algorithme efficace est un problème linéaire. Dans ILOG Hybrid [ILO02b], une contrainte globale intègre l'ensemble des contraintes linéaires. Après chaque propagation et modification de domaine de variables, la méthode du simplexe est utilisée sur cette sous-partie afin d'obtenir des bornes sur le coût ou sur certaines de ces variables (voir aussi [BK98]). Les contraintes ne faisant pas partie de cette sous-partie linéaire peuvent être de n'importe quel type admis par la programmation par contraintes. Celles-ci sont résolues en utilisant l'instantiation et la propagation.

### Plus-court chemin en programmation par contraintes

La même méthode peut être appliquée sur la partie du problème correspondant à un problème de plus court chemin avec fenêtres de temps. Il est alors possible d'exécuter de manière limitée, et si possible itérative, un algorithme à base d'étiquettes, qui pourra fournir des bornes sur les accumulations de ressources, des bornes sur le coût, et même des déductions logiques sur certains arcs impossibles. Nous revenons plus en détail sur des exemples de fonctionnement de la propagation d'une contrainte globale dans la section suivante. Dans [JKK<sup>+</sup>99], un tel schéma a été proposé, mais se limitant à une contrainte globale par ressource utilisant un algorithme de plus court chemin plus simple.

Il est évident que la complexité de l'algorithme *embarqué* dans la contrainte globale étant très élevée, il n'est pas possible de propager sur cette contrainte de manière systématique. De nombreuses limitations devront être utilisées afin de trouver de bons compromis entre le temps de propagation et la qualité des déductions. Ceci est vrai pour toute contrainte globale pouvant fournir un filtrage variant d'une simple propagation de bornes jusqu'à la résolution complète de la sous-partie du problème.

### Contrainte globale contraignant le plus court chemin avec fenêtres de temps

Nous proposons donc d'utiliser une contrainte globale dans un contexte de programmation par contraintes qui encapsule un filtrage utilisant l'algorithme d'étiquettes de SPRCTW. Le problème peut être défini dans l'environnement de programmation par contraintes ILOG Solver en utilisant la modélisation habituelle :

- des variables `next` définissent le chemin,
- des variables `cumul`, une distance et une contrainte `pathlength` définissent chaque ressource.

Le modèle peut être ensuite résolu en utilisant différents niveaux de propagation tels que définis dans la section 2.4.4 :

- une simple propagation de bornes, comme cela est fait par défaut dans ILOG Solver,
- un filtrage de type plus court chemin indépendant pour chaque contrainte de type `pathlength` présente dans le modèle, comme cela est proposé dans [JKK<sup>+</sup>99],
- un filtrage tenant compte de l'ensemble des ressources.

Notre proposition correspond à cette dernière possibilité. Une contrainte globale se charge alors de collecter l'ensemble des éléments compatibles avec l'algorithme de plus court chemin pur avec étiquettes. Ces éléments du modèle qui forment la partie pure compatible directement avec l'algorithme de filtrage sont de différents types. Ces éléments sont résumés dans la table 6.1.

Description	Exemple
Graphe	<code>IloIntArray nexts(env, nbVisits+2, 0, nbVisits+1);</code>
Ressource	<code>IloIntArray length(env, nbVisits+2, min, max); model.add(IloPathLength(env, nexts, length, LengthFunc));</code>
Objectif	<code>model.add(IloMinimize(env, objExpr));</code>
Expression obj.	<code>1 + length[nbVisits] + dist[nbVisits] - duals[nbVisits] IloMax(expr1, expr2);</code>
Borne sur obj.	<code>objExpr &lt;= maxObj</code>
Réductions graphe	<code>model.add(nexts[i] != j); model.add(min &lt;= nexts[i] &lt;= max);</code>
Fenêtres de temps	<code>model.add(min &lt;= length[i] &lt;= max);</code>

TAB. 6.1 – Types d'éléments collectables

En sortie, non seulement la contrainte globale peut fournir un filtrage adapté au problème, i.e. une réduction des domaines de certaines variables en fonction du domaine de certaines autres, mais elle peut également fournir une information heuristique. La réduction de domaines correspond principalement à l'élimination d'arcs et l'amélioration des bornes inférieures des domaines des variables représentant l'accumulation de ressources à certains nœuds. L'information heuristique est obtenue sous forme de *goals* proposant une heuristique d'instantiation du chemin. Plusieurs goals sont alors proposés :

- un goal instantiant directement le meilleur chemin selon l'algorithme à base d'étiquettes, sans donner la main au filtrage d'éventuelles autres contraintes.
- un goal instantiant directement les  $n$  meilleurs chemins correspondant aux  $n$  étiquettes présentes au nœud de fin de chemin.
- un goal identique au premier, mais laissant, après l'instantiation de chaque arc, la possibilité à d'autres contraintes de se propager et éventuellement

d'échouer.

**Exemples de propagation** L'utilisation d'un algorithme de filtrage plus complexe dans une contrainte globale peut comporter certaines difficultés de compréhension. Nous illustrons ici les interactions entre le système de programmation par contraintes et la contrainte globale permettant d'orienter et de réduire la recherche sur deux problèmes proches. Seul le deuxième problème comporte des contraintes additionnelles.

Supposons que nous avons le graphe représenté dans la figure 6.5. Pour chaque arc, les coûts associés sont ceux donnés dans la figure. D'autre part, une ressource est utilisée dont la consommation totale est limitée à 6 et les consommations sont données dans la table 6.2.

Arc	Consommation
A-B	1
A-C	1
A-D	3
B-C	1
B-F	2
C-E	2
C-F	2
D-E	2
E-F	3

TAB. 6.2 – Consommations de ressource

Imaginons tout d'abord que le problème est *pur*, i.e. qu'il ne contient que des contraintes compatibles avec l'algorithme de plus court-chemin. Cet algorithme, non inclus dans un environnement de programmation par contraintes, fournirait alors la solution optimale directement. A un nœud de l'arbre de recherche de la programmation par contraintes, cet algorithme incorporé dans une contrainte globale, fournit alors la solution optimale au problème tel qu'il est en tenant compte de toutes les décisions et déductions propagées du nœud racine jusqu'à ce nœud. Nous pouvons voir avec la figure 6.6 de quelle manière la contrainte globale le rend utilisable simplement dans le cadre d'une recherche arborescente. Les étapes pour arriver au résultat sont :

1. La contrainte globale propageant initialement nous indique que l'arc ( $A-D$ ) est impossible (en raison de la limitation de ressource) et que le meilleur chemin suivant l'état actuel est ( $A-C-F$ ),
2. la décision est prise d'utiliser l'arc ( $A-C$ ),

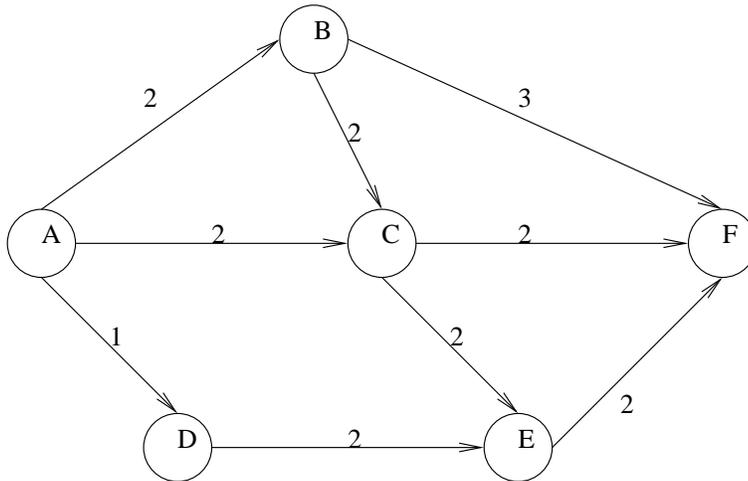


FIG. 6.5 – Graphe illustrant le fonctionnement de la contrainte globale

3. d'autres contraintes propagent alors et ne donnent aucun filtrage supplémentaire,
4. la décision est prise d'utiliser l'arc  $(C - F)$ ,
5. nous obtenons alors une solution de coût 4, la méthode du *branch-and-bound* consiste à continuer la recherche avec la décision contraire au dernier point de choix, et en obligeant le coût à être inférieur ou égal à 3,
6. en ajoutant la décision de ne pas utiliser  $(A - C)$ , l'algorithme de plus court chemin nous indique que le plus court chemin est de coût égal à 5, c'est un échec.

Au total, l'algorithme de filtrage a été invoqué 2 fois.

Imaginons maintenant que le problème n'est plus pur. Une contrainte non directement compatible avec l'algorithme de plus court chemin est ajoutée. Celle-ci interdit la séquence  $(A - C - F)$ . Ceci peut être formulé par :

```
if next[A]==C then next[C]!=F
```

Nous pouvons voir sur la figure 6.7 comment l'incorporation de l'algorithme dans une contrainte globale permet de traiter cette contrainte.

La séquence de recherche est alors la suivante :

1. de même qu'auparavant, la partie pure du problème indique que la meilleure décision à prendre est  $(A - C - F)$ , donc la première décision prise est d'utiliser l'arc  $(A - C)$ ,
2. la contrainte externe est alors autorisée à propager, et déduit que l'arc  $(C - F)$  n'est pas possible. Seule la décision C-E est alors possible. Les domaines étant modifiés, la contrainte de plus court chemin est alors également

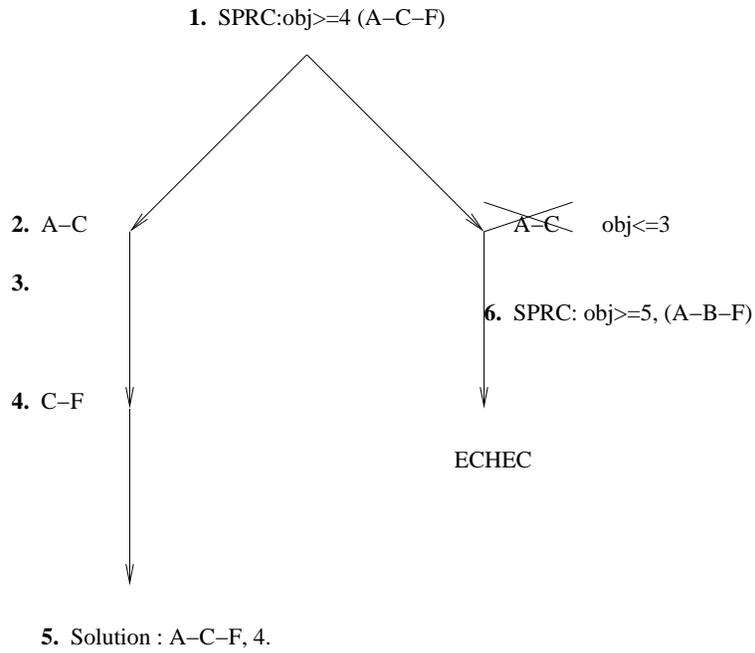


FIG. 6.6 – Exemple d’arbre de recherche pour un problème pur.

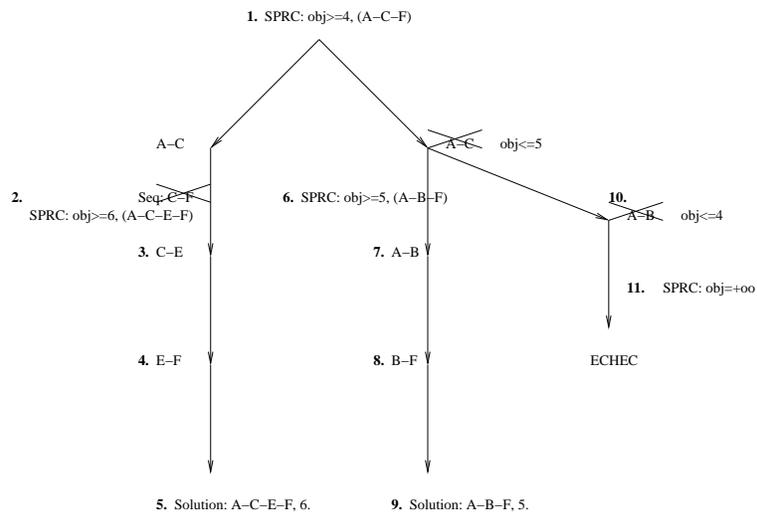


FIG. 6.7 – Exemple d’arbre de recherche pour un problème non pur.

*réveillée* et trouve le nouveau plus court chemin tenant compte des décisions prises. Ce chemin est  $(A - C - E - F)$  de coût 6.

3. l'arc  $(C - E)$  est alors utilisé,
4. puis l'arc  $(E - F)$ .
5. Une solution de coût 6 est trouvée. La méthode de *branch-and-bound* retourne au dernier point de choix avec la contrainte de coût inférieur ou égal à 5.
6. la contrainte de plus court chemin propose alors le chemin  $(A - B - F)$  de coût 5.
7. l'arc  $(A - B)$  est choisi,
8. l'arc  $(B - F)$  également,
9. une solution de coût 5 est trouvée.
10. une fois encore le *branch-and-bound* retourne au dernier point de choix.
11. la contrainte de plus court chemin ne trouve aucun chemin avec les décisions courantes, il ne peut donc pas exister de solution pour le problème complet, c'est un échec.

**Implantation** Deux particularités de notre implantation de cette contrainte globale dans l'environnement de programmation par contraintes ILOG Solver sont à signaler.

Tout d'abord, l'algorithme avec étiquettes n'est pas systématiquement exécuté sous sa forme complète. L'itérativité est mise à profit. Lorsqu'un choix ou une déduction d'une autre contrainte est faite, un domaine est réduit, i.e. un ou plusieurs arcs sont éliminés et la contrainte globale est notifiée. Les étiquettes utilisant les arcs éliminés sont alors elles-mêmes éliminées. Un ensemble réduit d'étiquettes pouvant donner une nouvelle prolongation peut ensuite être identifié. L'algorithme n'est utilisé que sur ces étiquettes. Au contraire, lors d'un backtrack d'une décision, aucune étiquette n'est supprimée mais toutes sont candidates à la prolongation. L'algorithme exécuté peut alors finir très rapidement si très peu d'étiquettes nouvelles sont créées. On voit donc par exemple que dans le cas d'un problème de plus court chemin pur, l'algorithme ne sera pas complètement exécuté deux fois.

C'est sous cette forme de contrainte globale que nous avons proposé l'algorithme de plus court chemin avec contraintes de ressources et fenêtres de temps dans notre environnement de génération de colonnes. En effet, comme nous venons de le voir, cette contrainte globale permet, quand le modèle ne contient que des contraintes compatibles avec l'algorithme de filtrage, de résoudre optimalement le problème sans échec et avec une seule exécution complète de l'algorithme. Concrètement, l'implantation comme contrainte globale oblige à prendre

en compte le temps nécessaire à l'actualisation des domaines des variables et d'appel d'actions quand ces domaines sont modifiés. L'expérience nous a montré que ce surplus de temps est souvent limité à moins de 10 %. Nos implantations sont décrites plus en détail dans l'annexe A.

### 6.4.3 Heuristique d'expert

Même si le graphe du sous-problème est acyclique, le nombre de combinaisons possibles au niveau des activités aériennes et des pairings est très élevé. La taille de l'arbre de recherche de programmation par contraintes correspondant est donc très grand, même en espérant une bonne propagation de la contrainte globale. Nous nous proposons alors de réduire heuristiquement cette recherche de manière à n'explorer que des parties correspondant à des pairings prometteurs. Avec la recherche arborescente de la programmation par contraintes, il est facile de concevoir cette limitation heuristique. A chaque nœud, le nombre de sous-branches explorées peut être limité. Si la recherche arborescente correspond à l'instantiation progressive d'un chemin, limiter le nombre de sous branches à un nœud correspond à limiter le nombre d'activités aériennes possibles après une activité fixée. L'utilisation de *goals* en programmation par contraintes permet de définir une telle heuristique de manière non déterministe, le choix s'effectuant lorsque toutes les propagations correspondant au chemin partiel ont été effectuées.

D'autre part, il ne faut pas oublier que la recherche de solution, avant l'utilisation d'un système informatique, est réalisée par une personne physique, n'étudiant qu'un nombre extrêmement restreint de combinaisons. Afin de choisir quelles combinaisons explorer, cette personne utilise intuitivement la confrontation de son expérience et des contraintes locales aux pairings. Cette combinaison résulte en fait en une heuristique leur permettant de décider intuitivement quelles sont les bonnes successions d'activités. Nous nous référons à cette heuristique comme l'*heuristique d'expert*. Nous pouvons extraire de ces méthodes intuitives des règles d'instantiation qui nous permettent de choisir l'activité suivante d'une manière proche de celle utilisée par l'expert. La première solution trouvée par notre recherche en programmation par contraintes correspond alors à la première prise en compte par l'expert (si celui ci ne viole pas de règle!).

### 6.4.4 Stratégies de recherche

Si nous utilisons une stratégie de recherche de type *Depth First Search* avec l'heuristique d'expert précédemment définie, beaucoup de temps peut être utilisé à rechercher dans une sous-partie de l'arbre correspondant aux mêmes  $n$  premières décisions de l'expert. Cependant, certaines des feuilles explorées correspondront à des décisions très différentes de celles qu'aurait prises l'expert. Si au contraire

nous utilisons une stratégie de recherche de type *Best First Search*, le nombre de décisions conformes et contraires à l'expert sera très varié.

En utilisant l'heuristique d'expert, nous faisons confiance au critère de l'expert pour prendre, en moyenne, de bonnes décisions, il serait donc logique d'étudier en priorité les parties de l'arbre de recherche correspondant au maximum aux décisions de l'expert. Dit sous une autre forme, nous voulons chercher dans les parties de l'arbre contenant le nombre minimum de choix différents de ceux qu'aurait réalisés l'expert. C'est ce que permet une stratégie de recherche du type *LDS* (*Limited Discrepancy Search*). Cette stratégie de recherche est illustrée dans les figures 6.8, 6.9 et 6.10. Chacune montre les nœuds de l'arbre de recherche visités pour une largeur donnée de la stratégie. Pour une largeur donnée, seule une petite partie de l'arbre de recherche est visitée. Contrairement à une stratégie de type DFS où tous les nœuds de la partie visitée correspondent aux mêmes décisions initiales (éventuellement mauvaises), ici, le paramètre de largeur indique le nombre d'*erreurs* (ou *différences*) acceptées, celles-ci pouvant être faites à n'importe quel niveau de la recherche.

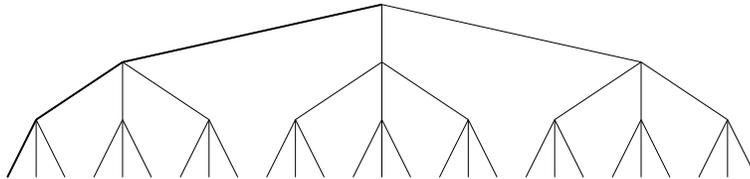


FIG. 6.8 – LDS avec largeur max. 1

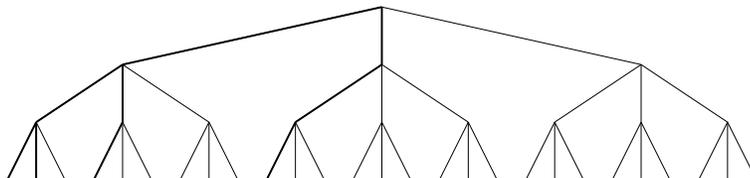


FIG. 6.9 – LDS avec largeur max. 2

### 6.4.5 Résolution du Problème Maître

Les heuristiques d'expert et stratégies de recherche que nous venons de proposer sont ensuite combinées dans l'environnement de programmation par contraintes. Le générateur obtenu est heuristique, i.e. des colonnes de coût réduit favorable peuvent ne pas être trouvées tant que la largeur du LDS utilisée est

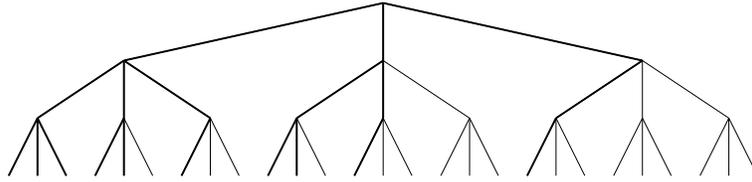


FIG. 6.10 – LDS avec largeur max. 3

inférieure à la somme des tailles des domaines des variables utilisées. Dans la pratique, cette largeur est inaccessible dans des temps acceptables et le générateur est donc utilisé de manière heuristique.

La résolution du problème maître peut donc être également réduite heuristiquement. L'optimum relâché ne pouvant être atteint, pour les raisons données auparavant, une recherche de type *génération de colonnes simple* est à exclure. Nous proposons donc de chercher des solutions entières en utilisant une recherche de type PLNE sur les colonnes courantes après chaque génération où des colonnes ont été trouvées.

La recherche peut donc être donnée par le goal :

```

HeurGener(int LDS) {
    ColonneArray colonnes;
    double[] duals = getDuals();
    trouveNouvellesColonnesLDS(colonnes, duals);

    if (colonnes.getSize())
        return IloAnd(colonnes, SolveLocalMIP(), this);

    return HeurGener(LDS+1);
}

```

Cette procédure peut être décrite de la manière suivante. Un niveau maximal de LDS devant être utilisé pour les recherches dans le sous-problème est utilisé. Si des colonnes sont trouvées en utilisant cette largeur de LDS, elles sont ajoutées, une solution entière est recherchée en utilisant la PLNE localement sur les colonnes existantes, et nous revenons à la génération avec la même largeur. Quand aucune colonne n'est trouvée, la largeur du LDS est augmentée. L'algorithme est arrêté quand une limite sur le temps d'exécution est atteinte.

## 6.5 Résultats

Deux problèmes différents ont été utilisés pour illustrer les deux types de modifications portant sur les schémas de coûts et les contraintes additionnelles, que nous nommons respectivement *Crew* et *Viva*.

Le premier problème utilise un modèle et des jeux de données simplifiés provenant d'une compagnie américaine. Il a été initialement réalisé dans le but d'illustrer le traitement possible des différents schémas de coûts.

Le deuxième problème correspond à un prototype réalisé pour traiter des problèmes de *génération de pairings* dans une compagnie aérienne espagnole. Les contraintes prises en compte ainsi que les tailles de données traitées sont proches de la réalité. Des contraintes additionnelles non compatibles avec l'algorithme à base d'étiquettes sont à prendre en compte pour limiter les temps de repos après des activités aériennes nécessitant l'utilisation d'un vol de situation.

Dans les deux cas, le problème comporte des contraintes sur les bases. Chaque fois, elles sont intégrées en utilisant différents générateurs pour chaque base.

### 6.5.1 Crew

Ce problème avait pour objectif de valider la modification de l'algorithme à base d'étiquettes dans le cas où des schémas de coûts différents sont utilisés. Pour ce problème, le coût utilisé est formé du maximum entre trois termes :

- une constante,
- le nombre d'heures de vol total (sommé sur les activités aériennes) multiplié par un coefficient,
- la durée totale entre le début et la fin du pairing, multipliée par un coefficient.

Dans la table 6.3 sont donnés les résultats obtenus. Nous comparons notre approche utilisant l'algorithme de plus court chemin modifié et embarqué dans une contrainte globale (ligne *SP*), avec plusieurs autres solutions. La ligne *S* correspond à un modèle de programmation par contraintes sans contrainte globale, la ligne *SH* correspond au même modèle utilisant une heuristique sur les sous-problèmes et le problème maître. Enfin, la ligne *Path* utilise un autre modèle de programmation par contraintes utilisant la formulation de chemin (les variables *next*), mais utilisant un niveau de filtrage inférieur, comme cela est décrit dans la section 2.4.4

### 6.5.2 Viva

Ce deuxième problème correspond à un prototype réalisé pour résoudre des problèmes réels de deux compagnies aériennes espagnoles. L'objectif initial était

Algorithmes	56 vols		124 vols		173 vols	
	OPT	Temps	OPT	Temps	OPT	Temps
<i>SH</i>	2403	4,2	4626	12,96	6164	55,47
<i>S</i>	2403	5,78	4625	31,31	6164	179
<i>Path</i>	2403	1,77	4625	46,7	6164	273,5
<i>SP</i>	2403	0.5	4625	2,9	6164	5,9

TAB. 6.3 – Résultats sur Crew

de démontrer la viabilité de la génération automatique de pairings d'une qualité comparable à celle obtenue par une personne expérimentée. Un problème type a donc été extrait du problème réel en éliminant certaines contraintes dont l'intérêt dans le cadre de ce prototype était limité. Celui-ci inclut la quasi totalité des contraintes provenant de la réglementation nationale espagnole. De nombreux contacts avec l'expert ont permis de développer les heuristiques utilisées selon la méthode proposée dans la section 6.4.3.

### Commentaires sur les jeux de données

Nos essais ont été réalisés sur trois jeux de données différents. Ceux-ci proviennent de deux compagnies aériennes différentes, notées *V* et *A*, comportant respectivement 9 et 19 avions. Pour la première compagnie, nous disposons de deux jeux de données, correspondant à deux mois différents.

### Résultats

	VA10	VM31	A11
Nombre d'avions	9	9	19
Nombre de jours	10	31	11
Nombre de vols	526	975	1130
Nombre d'activités	662	1092	3154
Solution de l'expert	212		
Temps pour égaler l'expert	33 s.		
Meilleure solution	197	428	432
Gain (%)	6,2		
Temps pour meilleure solution	172 s.	329 s.	28 min.

TAB. 6.4 – Meilleurs résultats obtenus.

Les résultats obtenus avec le prototype réalisé sont donnés dans la table 6.4.

L'objectif numérique initial était d'obtenir un résultat utilisant un maximum d'un pilote de plus que l'expert par jour de planification. Cela nous permettait, dans le cas de *VA10*, d'obtenir un résultat de  $212 + 10 = 222$ . Le résultat obtenu est meilleur de presque deux pilotes par jour. Malheureusement, pour les autres instances, nous ne disposons d'aucun résultat obtenus par un expert, ce qui nous interdit de réaliser le même type de comparaison.

Enfin, nous donnons dans la figure 6.11 une représentation de l'amélioration de la qualité des solutions obtenues en fonction du temps pour plusieurs stratégies. La meilleure d'entre elles (*FixedRC0*) utilise des déductions sur des variables en fonction du coût réduit.

## 6.6 Conclusions

Dans ce chapitre, nous avons présenté une application des méthodes de génération de colonnes aux problèmes de planification de ressources en utilisant comme exemple le problème de génération de *pairings*. Le problème de plus court chemin apparaissant dans le sous-problème est plus complexe que dans le cas du VRP du chapitre précédent et fait intervenir de nouveaux types de contraintes. Nous avons alors proposé plusieurs contributions permettant d'améliorer la résolution de ce sous-problème. Tout d'abord, nous avons montré comment certaines des contraintes pouvaient être traitées avec l'algorithme habituel d'étiquettes en modifiant légèrement celui-ci. Ensuite, nous avons montré comment cet algorithme pouvait être formulé dans un environnement de programmation par contraintes lorsque les contraintes additionnelles ne sont pas directement intégrables. Le modèle de contrainte globale proposé utilise alors un niveau de filtrage plus élevé que dans les études précédentes. Nous proposons également une modification heuristique et méta-heuristique de la recherche.

Ces contributions sont alors validées sur un problème réel et les résultats sont comparés favorablement avec ceux obtenus par un expert réalisant le travail manuellement. Il n'existe malheureusement pas de benchmark réaliste public pour ce type de problème sur lequel nous pourrions comparer notre approche avec d'autres méthodes.

Pour les deux problèmes présentés jusqu'à maintenant, nous avons centré nos efforts sur la résolution du sous-problème. Dans le prochain chapitre, nous introduirons une nouvelle application des méthodes de génération de colonnes et de coupes avec sous-problème de plus court chemin. Nous proposerons alors plusieurs contributions portant sur le traitement du problème maître.

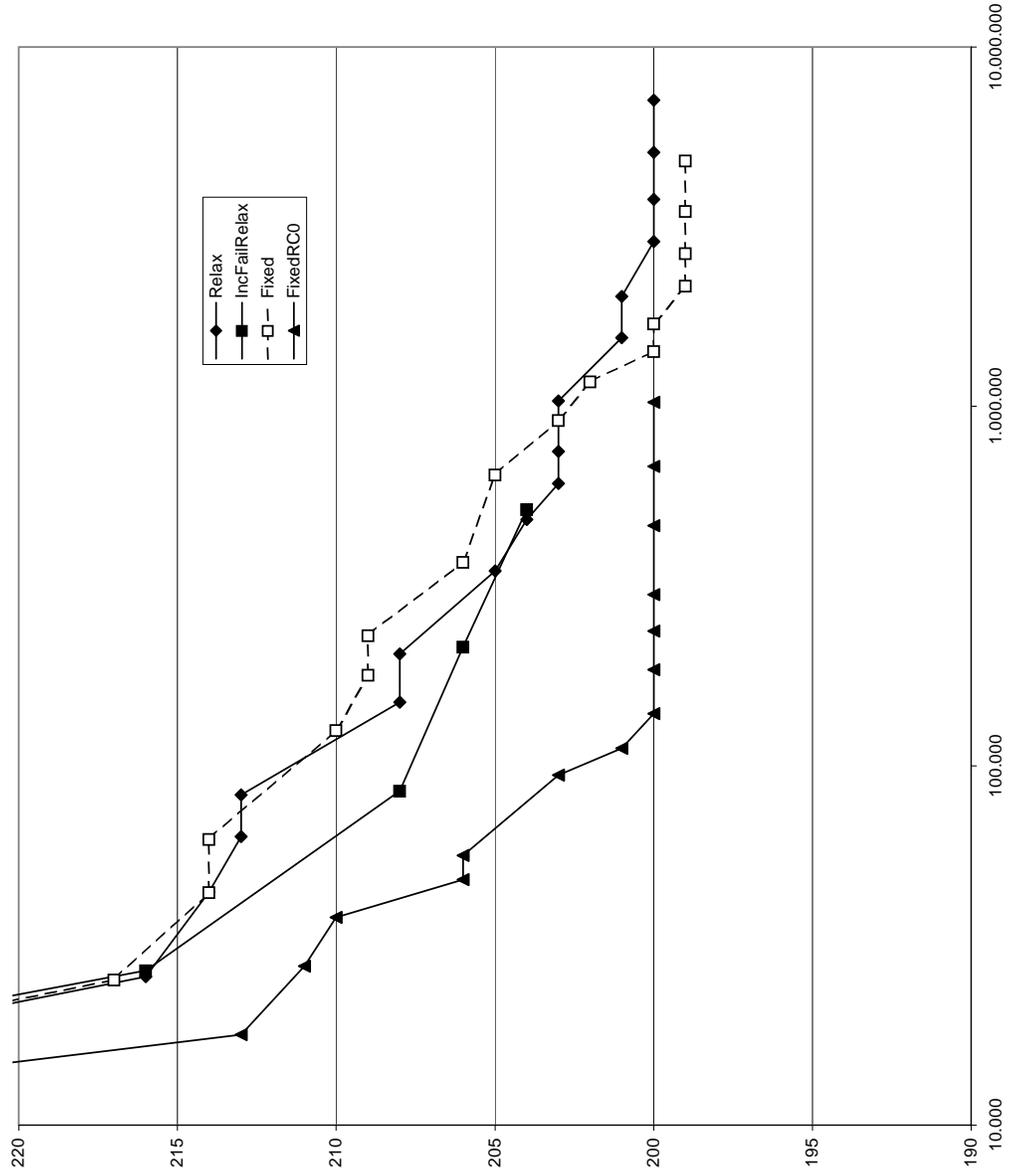


FIG. 6.11 – Evolutions des résultats dans le temps



## Chapitre 7

# Coupes, heuristiques et stratégies de recherche dans le problème maître : application à la conception de réseau.

Dans les deux chapitres précédents, nous avons proposé diverses méthodes permettant une résolution plus efficace des sous-problèmes de plus court chemin. Cependant, la résolution du sous-problème ne comporte parfois pas de difficulté particulière. De très nombreuses colonnes peuvent alors facilement être générées, ce qui peut donner lieu à de mauvaises bornes inférieures et supérieures. Ce chapitre présente certaines modifications pouvant alors être apportées au traitement du problème maître ainsi qu'à la procédure de recherche globale. En particulier, nous proposons d'utiliser des coupes tenant compte des structures internes des colonnes ainsi que des réductions heuristiques et stratégies de recherche du problème maître. Ces modifications sont appliquées à la résolution d'un problème de conception de réseau. Nous montrons qu'alors la *génération de colonnes et de coupes heuristique* peut être efficace et robuste pour trouver des solutions rapidement.

Le *Problème de Conception de Réseau* recouvre une famille importante de problèmes consistant à dimensionner les arcs d'un réseau de télécommunications afin que plusieurs commodités puissent être simultanément routées sur le réseau sans excéder les capacités des arcs choisis. Sur chaque arc, la capacité pouvant être installée peut être choisie parmi un domaine discret. A chaque niveau de capacité possible correspond un coût différent. L'objectif est de minimiser le coût total de construction du réseau.

Différents aspects du problème de conception de réseau ont déjà été étudiés. Chaque étude se centre cependant sur un cas précis et utilise un modèle limité

sur lequel sont développées des techniques spécifiques. Dans les applications industrielles, une préoccupation importante est la robustesse. L'utilisateur typique attend de l'algorithme qu'il trouve de bonnes solutions non seulement sur une sous classe particulière de problèmes, mais aussi qu'il donne des solutions de qualité acceptable quand des contraintes supplémentaires sont ajoutées au problème. Les études que nous avons mentionnées auparavant résolvent généralement des problèmes provenant de benchmark académiques qui ne reflètent pas cette préoccupation.

Dans ce chapitre, nous présentons une approche de *Génération de Colonnes et de Coupes heuristique* qui s'est avérée robuste sur un ensemble étendu de benchmarks provenant du projet ROCOCO et qui ont été construits sur la base de problèmes de conception de réseaux utilisant des données réelles fournies par France Télécom [BCP<sup>+</sup>02]. Le benchmark inclut trois séries différentes contenant chacune jusqu'à sept instances. Aussi, six classes de contraintes additionnelles sont définies, donnant un total de 64 configurations différentes pour chacune des instances. Initialement, l'objectif était d'obtenir un algorithme offrant de bonnes solutions et bornes inférieures en moyenne pour ces  $3 \times 7 \times 64 = 1344$  instances en un temps limité à 10 minutes par configuration. En réalité, la plupart des participants au projet se sont concentrés sur la recherche de bornes supérieures. Un large éventail de techniques différentes a déjà été testé sur ce challenge : Programmation par Contraintes, Recherche Locale, Programmation Linéaire, etc. Jusqu'à maintenant, les meilleurs résultats publiés étaient obtenus en utilisant un algorithme hybridant la Programmation par Contraintes et la Recherche Locale. Cet algorithme est présenté dans [PPRS02]. Au début, il semblait peu probable que des techniques de génération de colonnes puissent être compétitives car elles sont en général considérées peu efficaces pour obtenir rapidement de bonnes solutions, et en particulier lorsqu'elles sont comparées aux méthodes de Recherche Locale. Notre algorithme non seulement est capable de donner de bonnes bornes inférieures (en particulier grâce à l'ajout de coupes) mais les améliorations proposées le rendent également compétitif pour obtenir de bonnes solutions rapidement (en particulier grâce aux stratégies de recherche).

Ce chapitre est organisé de la manière suivante. Dans la section 7.1, nous présentons le problème de conception de réseau et en donnons une formulation complète. Ensuite, dans les trois sections suivantes, nous présentons les différents aspects de notre approche de *Génération de Colonnes et de Coupes heuristique*. Dans la section 7.2, le modèle décomposé est introduit ainsi que les différents détails des techniques de génération de colonnes utilisés. Puis, dans la section 7.3, nous introduisons les coupes que nous avons utilisées pour améliorer les bornes inférieures obtenues à chaque nœud et montrons comment nous les utilisons pour réduire l'arbre de recherche. Enfin, dans la section 7.4, nous présentons les heuristiques et stratégies de recherche que nous utilisons pour permettre d'obtenir

rapidement des solutions de bonne qualité. Enfin, dans la section 7.5, nous donnons une synthèse de nos résultats et les comparons à d'autres résultats publiés.

Les résultats présentés dans ce chapitre ont été en partie obtenus lors de travaux réalisés dans le cadre du projet RNRT ROCOCO financé par le MENRT français.

## 7.1 Le benchmark de conception de réseau

Les problèmes de conception de réseau que nous traitons dans ce chapitre font partie du benchmark compilé dans le cadre du projet ROCOCO et sont décrits en détail dans [BCP<sup>+</sup>02] et [PPRS02]. Ces problèmes correspondent à une des deux grandes variantes des problèmes de conception de réseau où chaque commodité doit suivre un chemin unique. Le benchmark complet a été construit à partir de données réelles fournies par France Télécom et forme un ensemble d'un total de 1344 instances.

Nous reprenons ici la description du problème de base tel qu'il est donné dans [PPRS02] dont nous reprenons une grande partie des notations.

Soit un graphe  $G = (X, A)$ , avec  $X$  un ensemble de  $N$  nœuds, et  $A$  un ensemble d'arcs entre ces nœuds.

Un ensemble de demandes est également défini. Chaque demande associe à une paire  $(p, q)$  une quantité entière  $Dem_{pq}$  de données devant être routée selon un chemin unique de  $p$  à  $q$ . En principe, il peut y avoir plusieurs demandes pour la même paire  $(p, q)$ . Dans ce cas chaque demande peut être routée selon un chemin différent. Comme une seule instance du benchmark (C20) possède cette propriété, nous ne la considérerons pas afin de simplifier la présentation des résultats au cas où à une paire  $(p, q)$  correspond une demande unique  $Dem_{pq}$ . Ceci rend les résultats plus simples à présenter, chacun d'entre eux étant facilement généralisable au cas où plusieurs demandes sont associées à une paire origine-destination.

Pour chaque arc  $(i, j)$ ,  $K_{ij}$  capacités possibles  $Capa_{ij}^k$ ,  $1 \leq k \leq K_{ij}$ , sont données. Au plus une de ces  $K_{ij}$  capacités peut être choisie. Cependant, il est permis de multiplier cette capacité par un entier inclus dans un intervalle  $[Wmin_{ij}^k, Wmax_{ij}^k]$ . Le problème consiste alors à sélectionner pour certains arcs  $(i, j)$  une capacité  $Capa_{ij}^k$  et un multiplicateur  $w_{ij}^k$ . Ces choix pour les arcs  $(i, j)$  et  $(j, i)$  sont liés. Si la capacité  $Capa_{ij}^k$  est choisie pour l'arc  $(i, j)$  avec le multiplicateur  $w_{ij}^k$ , alors la capacité  $Capa_{ji}^k$  doit être choisie pour l'arc  $(j, i)$  avec le même multiplicateur  $w_{ji}^k = w_{ij}^k$ , et avec un coût pour l'ensemble des deux arcs  $(i, j)$  et  $(j, i)$  de  $w_{ij}^k * Cost_{ij}^k$ . Aucune demande ne peut transiter par les arcs pour lesquels aucune capacité n'a été choisie.

Sur ce modèle commun, six classes de contraintes additionnelles sont définies.

Pour une instance donnée, il y a donc 64 configurations possibles de problèmes, indexées en utilisant un vecteur de six bits.

- La contrainte de sécurité (*sec*) oblige certaines demandes à être sécurisées. Pour chaque nœud  $i$ , un indicateur  $Risk_i$  indique si le nœud est considéré à risque. De manière similaire, pour chaque arc  $(i, j)$  et chaque niveau  $k$ ,  $1 \leq k \leq K_{ij}$ , un indicateur  $Risk_{ij}^k$  définit si le lien  $(i, j)$  avec le niveau de capacité  $k$  est risqué. Une demande devant être sécurisée ne peut pas être routée à travers un de ces éléments à risque.
- La contrainte de non multiplication des capacités (*nomult*) interdit l'utilisation des multiplicateurs de capacité.
- La contrainte de routage symétrique des demandes symétriques (*symdem*) oblige que pour une paire  $(p, q)$ , toutes les demandes de  $p$  à  $q$  soient routées selon le même chemin, et toutes les demandes de  $q$  à  $p$  soient routées selon le chemin symétrique.
- La contrainte de nombre maximum de bonds (*bmax*) oblige chaque route utilisée pour acheminer une demande à avoir un nombre maximum de bonds (i.e. arcs) spécifique pour chaque demande  $Bmax_{pq}$ .
- La contrainte de nombre maximum de ports (*pmax*) associe à chaque nœud  $i$ , une limite  $Pin_i$  sur le nombre de ports entrants, et une limite  $Pout_i$  sur le nombre de ports sortants. Pour chaque nœud, nous devons alors avoir  $\sum_{j,k} w_{ij}^k \leq Pout_i$  et  $\sum_{j,k} w_{ji}^k \leq Pin_i$ .
- La contrainte de trafic maximum (*tmax*) associe à chaque nœud  $i$  une limite  $Tmax_i$  sur le trafic routé à travers ce nœud. Ceci inclut le trafic qui commence en  $i$ , le trafic qui finit en  $i$ , et le trafic qui transite par  $i$ .

Dans certaines parties de ce travail, nous utilisons une particularité de certains de nos jeux de données, en particulier pour certains résultats sur les coupes. Sur ces instances, les capacités sont identiques pour tous les arcs et sont toutes multiples d'une capacité de base *CapaDiv*. Chaque capacité est alors donnée par :

$$Capa_{ij}^k = k.CapaDiv \quad (7.1)$$

Vingt-deux fichiers de données, organisés en trois séries, sont disponibles. Chaque fichier est identifié par une lettre correspondant à sa série (A, B ou C) et un entier indiquant le nombre de nœuds dans le réseau utilisé. La série A inclut les instances les plus petites (de 4 à 10 nœuds), qui ont été historiquement les premières étudiées et incluses dans le benchmark. Ensuite ont été ajoutées les séries B et C qui contiennent des problèmes de taille plus importante (de 10 à 25 nœuds).

Finalement, mentionnons que ce benchmark est public et que les instances sont disponibles à <http://www.prism.uvsq.fr/Rococo>. Dans [BCP<sup>+</sup>02], une description complète des formats utilisés pour les fichiers et des caractéristiques des différentes instances est donnée.

Ce benchmark est particulièrement intéressant car il recouvre de nombreuses situations correspondant à des problèmes étudiés auparavant. Le problème de base correspond en effet à un problème de flux binaire avec multiples commodités et multiples capacités. Ensuite les 6 catégories de contraintes additionnelles essaient de correspondre aux multiples variantes apparaissant dans des applications industrielles. L'objectif n'est alors pas d'avoir l'algorithme le plus efficace uniquement sur le problème de base, mais celui qui permet d'obtenir en moyenne la meilleure qualité de bornes inférieures et supérieures sur l'ensemble des configurations en donnant un maximum de 10 minutes à la résolution de chacune. Un grand nombre d'autres techniques d'optimisation ont été testées sur ce benchmark. Certains résultats sont donnés dans [PPRS02] et [BCP<sup>+</sup>02].

Nous proposons de résoudre de manière robuste ce challenge en utilisant des techniques de génération de colonnes sur un modèle décomposé comportant un sous-problème de plus court chemin.

## 7.2 Modèles de Génération de Colonnes

Dans cette section, nous présentons les différents modèles possibles pour ce problème. Nous donnons tout d'abord un modèle PLNE sur lequel des méthodes de génération de colonnes pourraient être (et ont été dans certaines études) utilisées. Nous présentons ensuite un modèle décomposé.

Pour rendre plus faciles les explications, nous ne présenterons ici que le modèle correspondant à la configuration 010000 des contraintes additionnelles. Nous reviendrons plus tard sur la manière dont chacune des classes de contraintes additionnelles modifie ce modèle.

### 7.2.1 Modèle PLNE

Dans [Gun98], un modèle simple de PLNE est utilisé pour un problème similaire de conception de réseau avec capacité. Pour chaque demande  $(p, q) \in X^2$  et chaque arc  $(i, j) \in X^2$  une variable binaire  $x_{ij}^{pq}$  est définie qui détermine si la demande  $(p, q)$  est routée à travers l'arc  $(i, j)$ . Aussi, pour chaque arc  $(i, j)$ , et chaque niveau de capacité  $k$ , une variable binaire  $y_{ij}^k$  indique si le lien de niveau

de capacité  $k$  est ouvert entre les nœuds  $i$  et  $j$ . Le modèle résultant est :

$$\text{minimise } \sum_{i,j} \sum_{k=1}^{K_{ij}} Cost_{ij}^k y_{ij}^k \quad (7.2)$$

$$\sum_j x_{pj}^{pq} = 1, \forall(p, q) \quad (7.3)$$

$$\sum_i x_{iq}^{pq} = 1, \forall(p, q) \quad (7.4)$$

$$\sum_i x_{in}^{pq} = \sum_i x_{ni}^{pq}, \forall(p, q, n) \quad (7.5)$$

$$\sum_{k=1}^{K_{ij}} y_{ij}^k \leq 1, \forall(i, j) \quad (7.6)$$

$$\sum_{(p,q)} Dem_{pq} x_{ij}^{pq} \leq \sum_{k=1}^{K_{ij}} Capa_{ij}^k y_{ij}^k, \forall(i, j) \quad (7.7)$$

$$x_{ij}^{pq} \in \{0, 1\}$$

$$y_{ij}^k \in \{0, 1\}$$

Les équations (7.3), (7.4) et (7.5) assurent qu'un unique chemin existe entre  $p$  et  $q$ . Les équations (7.6) assurent qu'au plus un type de lien est utilisé sur un arc  $(i, j)$ . Finalement, les équations (7.7) assurent que la limite de capacité sur l'arc  $(i, j)$  est respectée. Ce modèle a été testé sur le benchmark du projet ROCOCO. Certains résultats et discussions à ce sujet sont donnés dans [PPRS02]. Même si la méthode peut donner des résultats satisfaisants, notre sentiment est que si de nombreuses demandes sont définies, le nombre de variables va exploser.

Des techniques de génération de colonnes peuvent être utilisées sur ce modèle. En effet, il est possible de ne pas inclure toutes les variables  $x$  et  $y$  dans le modèle initial mais de les ajouter durant la recherche quand cela nécessaire. Ceci est proposé par exemple dans [OW00] et [CG95]. Les sous-problèmes mis en jeu sont très simples, car énumératifs. Aucun problème de plus court chemin n'intervient.

Nous proposons de décomposer le problème et d'obtenir un modèle alternatif sur lequel nous appliquerons des méthodes de génération de colonnes de manière efficace. Ce modèle fera intervenir des sous-problèmes de plus court chemin.

## 7.2.2 Modèle décomposé

Le programme linéaire en nombres entiers est décomposé en utilisant la décomposition de Dantzig-Wolfe, comme dans la section 1.2.4. La décomposition est

appliquée ici sur les variables  $x$  dont les points extrêmes correspondent aux chemins valides pour la demande correspondante. Nous n'allons pas, cette fois ci, présenter la décomposition en détail, celle-ci s'effectuant de manière identique aux applications précédentes. Nous décrivons simplement le modèle décomposé résultant qui fait intervenir les deux types de colonnes suivantes :

- colonnes de type *Path*, représentant un chemin réalisable pour une demande  $(p, q)$ , et défini par une liste ordonnée des nœuds traversés. Les variables correspondantes sont des variables binaires. Nous noterons  $Path_{pq}$  l'ensemble des colonnes possibles pour la demande  $(p, q)$ .
- colonnes de type *Link*, correspondant à un lien possible pour un arc  $(i, j)$  et un niveau de capacité  $k$ . Elles sont directement équivalentes aux variables  $y$  du modèle d'origine qui n'ont pas été décomposées. Les variables correspondantes sont binaires. Nous noterons  $Link_{ij}$  l'ensemble des colonnes de lien pour l'arc  $(i, j)$  et  $Link_{ij}^k$  la variable/colonne correspondant au lien possible entre  $i$  et  $j$  avec le niveau de capacité  $k$ .

Toutes les contraintes du problème de base peuvent être écrites comme des contraintes du problème maître suivant :

$$\text{minimise } \sum_{i,j} \sum_{k=1}^{K_{ij}} Cost_{ij}^k Link_{ij}^k \quad (7.8)$$

$$\sum_{x \in Path_{pq}} x = 1 \quad (7.9)$$

$$\sum_{k=1}^{K_{ij}} Link_{ij}^k \leq 1 \quad (7.10)$$

$$\sum_{pq} (Dem_{pq} \sum_{x \in Path_{pq}, (i,j) \in x} x) \leq \sum_{k=1}^{K_{ij}} Capa_{ij}^k Link_{ij}^k \quad (7.11)$$

$$Link_{ij}^k \in \{0, 1\} \quad (7.12)$$

$$x \in \{0, 1\}, \forall x \in Path_{pq}. \quad (7.13)$$

Les équations (7.9) forcent une demande  $(p, q)$  à ne correspondre qu'à un seul chemin. Les équations (7.10) forcent un arc  $(i, j)$  à n'être associé qu'à au plus un lien. Enfin, la contrainte de capacité (7.11) force la somme des demandes passant à travers un arc à être inférieure à la capacité totale disponible.

L'équation (7.10) est en fait  $\sum_{k=1}^{K_{ij}} Link_{ij}^k \in \{0, 1\}$ . Le cas correspondant à l'expression égale à 0 correspond au cas où aucun chemin ne peut passer sur cet arc.

Nous appliquons alors des techniques de génération de colonnes à ce modèle décomposé. Les sous-problèmes consistent à trouver les nouveaux chemins et liens

réalisables. Dans le problème de base, les liens sont directement tirés des données et les chemins sont les chemins élémentaires de  $p$  à  $q$ . Les contraintes additionnelles peuvent ensuite être intégrées sous forme de modification d'un de ces sous-problèmes.

### Modèle générique

Les colonnes sont dans ce problème de deux types différents. Nous introduirons alors deux variables d'état comme définies dans la section 3.1.3 :  $link$  et  $path$  qui prennent la valeur 1 dans leur sous-problème respectif et 0 dans l'autre. Dans le sous-problème décrivant les nouveaux liens, trois variables de décisions sont définies :  $link.k$ ,  $link.i$  et  $link.j$ . Dans le sous-problème pour des nouveaux chemins, sont également définies  $path.p$ ,  $path.q$  et  $path.next$  qui décrit le chemin comme dans les applications précédentes. Nous pouvons alors facilement définir des expressions de prix pour les contraintes maîtres de ce problème. Par exemple, une contrainte (7.9) utilise une expression de prix ne valant le coefficient 1 que pour les chemins allant de  $p$  à  $q$  :

$$CM_{pq} = ((1, 1), path.(p = path.p).(q = path.q)).$$

De même, une contrainte maître de (7.10) vaut :

$$CM_{ij} = ((-\infty, 1), link.(i = link.i).(j = link.j)).$$

Enfin, une contrainte de (7.11) :

$$CM_{ij}^2 = ((-\infty, 0), \\ Dem_{pq}.path.(path.next[i] = j) \\ -link.(i = link.i).(j = link.j)).Capa[link.i][link.j][link.k]).$$

### 7.2.3 Contraintes additionnelles

Nous présentons ici brièvement comment le modèle décomposé est modifié quand des contraintes additionnelles parmi les six classes définies sont ajoutées.

Deux contraintes sont extrêmement faciles à prendre en compte. La contrainte de routage symétrique est intégrée en ne prenant en compte que les colonnes pour  $(p, q)$  avec  $p < q$ , et en les utilisant pour les demandes dans les deux sens. Des ajustements très simples d'expressions de prix sont réalisés. La contrainte sur le nombre maximum de bonds limite également l'ensemble des chemins devant être pris en compte. Le sous-problème devient alors un problème de plus court chemin de taille limitée. La taille peut facilement être prise en compte sous forme d'une ressource. Ces deux contraintes rendent le problème complet plus petit.

Les contraintes de non multiplication changent simplement les données. En effet, l'existence de multiplicateurs peut être prise en compte en créant artificiellement de nouveaux niveaux de capacités.

Les contraintes sur le nombre maximum de ports et de trafic correspondent à de nouvelles contraintes maître simples. Par exemple, la contrainte sur le nombre maximum de ports sortant du nœud  $n$  s'écrirait :

$$CM_n = ((0, Pout_n), link.(n = link.i).w[link.i][link.j][link.k]).$$

De manière similaire, la contrainte sur le trafic maximum à travers le nœud  $n$  serait :

$$CM_n^2 = ((0, Tmax_n), path.(path.next[n] \neq n).dem[path.p][path.q]).$$

Enfin, même si les contraintes de sécurité peuvent être simplement modélisées sous forme de deux jeux de contraintes maîtres, leur difficulté correspond à leur grand nombre :  $NbDemandes * NbNoeuds$ . Même pour des petits problèmes de 10 nœuds, ceci peut être intraitable dans la pratique. Nous avons alors décidé de n'ajouter ces contraintes que quand cela est nécessaire durant la recherche, i.e. quand elles sont violées.

#### 7.2.4 Génération de Colonnes Simple

Si  $N$  est le nombre de nœuds, le nombre de contraintes dans le problème maître complet peut être grossièrement sous-estimé par  $N^2 + N^2 + N^2 = 3.N^2$  et le nombre de colonnes par  $N! + N^2$ . Avec  $N = 10$ , ceci correspond environ à 300 contraintes et 3 628 900 colonnes. Toutes les colonnes ne peuvent pas être ajoutées au modèle dès le début et des méthodes de génération de colonnes devront être utilisées. Le nombre de colonnes de type *Link* étant limité ( $N^2$ ), celles-ci seront toutes ajoutées au modèle dès l'initialisation. Une génération de colonnes retardée est utilisée pour les colonnes de type *Path*. Elles sont le résultat d'un sous-problème de plus court chemin.

La toute première méthode utilisée pour générer des colonnes de coût réduit favorable a simplement consisté à pré-calculer tous les chemins. Quand la génération de nouvelles colonnes est invoquée, le coût réduit de toutes celles non encore présentes dans le problème maître restreint est estimé. Celles dont le coût réduit est négatif sont alors ajoutées. Nous avons limité le nombre de colonnes entrant dans le problème à 50 par itération.

Une procédure de *génération de colonnes simple* telle que définie dans la section 2.2 a initialement été utilisée. Les résultats obtenus n'ont pas été satisfaisants. En effet, la table 7.1 montre que la solution entière trouvée (colonne *UB*) est loin de la borne inférieure donnée par la solution relâchée (colonne *LB*). Cette table

Instance	LB	OPT	UB	écart
A04	11622,49	22267	22267	47,80%
A05	17098,94	30744	30744	44,38%
A06	17793,17	37716	39821	55,32%
A07	24021,86	47728	50774	52,69%
A08	28182,29	56576	57466	50,96%

TAB. 7.1 – Résultats de la génération de colonnes simple sur la série A.

compare également la solution optimale connue (colonne *OPT*) avec notre *borne supérieure* (i.e. notre solution). De manière à obtenir de meilleures solutions sans changer trop ni le modèle ni la procédure, nous avons proposé diverses modifications telles que la possibilité d'ajouter des colonnes de coût réduit positif. En fonction du nombre de ces colonnes, l'exécution du PLNE peut être plus longue et est alors limitée suivant divers critères. Comme indiqué dans [PPRS02], ces modifications n'ont pas permis d'améliorer significativement les résultats obtenus avec cette procédure de génération de colonnes simple.

D'autre part, la table 7.1 montre également l'écart (colonne *écart*) entre les bornes inférieures et supérieures. Même si la solution obtenue n'est pas si lointaine de la solution optimale, l'écart entre les bornes est important.

### 7.2.5 *Branch-and-Price*

De manière à obtenir de meilleures solutions, éventuellement optimales, ou au moins réduire l'écart entre les bornes, une procédure de *Branch-and-Price* a ensuite été utilisée.

La principale difficulté, comme cela a été souligné auparavant, a alors été de trouver un schéma de branchement compatible avec le sous-problème. Nous présentons ici la stratégie et les diverses règles de branchement utilisées ainsi que la méthode de génération modifiée.

#### La stratégie et les règles de branchement

Après une première série d'essais, nous nous sommes concentrés sur deux types de règles de branchement, correspondant aux deux types de colonnes utilisés :

- les branchements de type *Arc* : pour une demande donnée  $(p, q)$  et un arc donné  $(i, j)$ , cette règle oblige l'arc à être utilisé ou non dans le chemin correspondant à la demande. Cette règle est identique à celle utilisée pour le problème de tournées de véhicules et correspond à la règle de Ryan-Foster.
- les branchement de type *Link* : pour un arc donné  $(i, j)$  et un niveau de

capacité  $k$ ,  $1 \leq k \leq K_{ij}$ , cette règle indique si le lien de niveau de capacité  $k$  est ouvert sur l'arc  $(i, j)$ .

Nos tests ont alors montré qu'une bonne stratégie consistait à brancher en priorité avec des règles de type *Link*, puis si nécessaire, (i.e. si toutes les variables de type *Link* sont entières), avec des règles de type *Arc*. Dans les deux cas, les règles choisies en priorité sont celles s'appliquant aux colonnes de plus hauts pseudo-coûts (plus grands  $|x - \text{round}(x)| * c$ , où  $c$  est le coût du lien ou la valeur de la demande correspondant au chemin).

### La méthode de génération

Nous générons toujours dynamiquement les colonnes de type *Path*. Les règles de type *Link* ne s'appliquant pas sur ces colonnes, il n'est pas nécessaire de modifier le modèle des sous-problèmes avant d'avoir atteint une profondeur de recherche où sont utilisées des règles de type *Arc*. Dans le cas où les colonnes de type *Path* sont générées en utilisant un algorithme de plus court chemin, les règles de type *Arc* sont facilement intégrées en modifiant légèrement le graphe, comme cela a été expliqué dans un chapitre précédent.

Nous utilisons alors plusieurs générateurs différents.

Le premier et plus simple générateur est adapté de celui utilisé pour la génération de colonnes simple. Tous les chemins de longueur inférieure à un seuil sont générés à l'initialisation de l'algorithme et ajoutés à une *réserve*. Quand la génération de nouvelles colonnes est invoquée, le coût réduit et la validité pour les règles de branchement des colonnes de cette réserve non encore ajoutées au problème maître sont estimés. Quand la valeur est négative, la colonne est ajoutée au nœud courant de l'arbre de recherche.

Quand aucune autre méthode plus rapide n'est utilisable, une autre générateur, extrêmement opposé, énumère récursivement tous les chemins possibles, tout en calculant leur coût réduit et en estimant leur faisabilité vis-à-vis des contraintes de base du sous-problème et des règles de branchement. Les chemins valides et de coût réduit négatif sont alors ajoutés au nœud courant. Ces chemins sont également ajoutés à la réserve de colonnes, ce qui permettra ultérieurement de les réutiliser facilement à un autre nœud.

L'idée derrière cette combinaison de deux générateurs via une réserve de colonnes est que la solution finale sera souvent composée d'un nombre important de colonnes provenant d'un petit sous-ensemble de colonnes qui peuvent être pré-calculées et sur lesquelles la validité pour un nœud et le coût réduit peuvent être rapidement estimés. Seules quelques autres devront être trouvées en utilisant des méthodes plus complexes, à la demande. Le coût correspondant à la recherche de celles-ci peut n'être compté qu'une fois, même si la colonne va être utilisée à de nombreux nœuds, en utilisant la réserve. Le sous ensemble des colonnes promet-

teuses a été construit en prenant les colonnes de longueur inférieure à un seuil donné, mais d'autres critères pourraient être envisagés.

Enfin, entre ces deux extrêmes, d'autres méthodes plus efficaces peuvent être utilisées dans des cas spécifiques. Par exemple, un algorithme de Ford est utilisé pour certaines configurations de contraintes additionnelles, quand celles-ci ne contiennent pas les contraintes *bmax*.

La procédure générale consiste alors à toujours privilégier la réserve, puis, si possible, à essayer un des générateurs spécifiques, et enfin en cas d'échec, à utiliser le générateur récursif.

## Résultats

Dans la table 7.2 sont donnés les résultats obtenus sur les plus petites instances de la série *A* avec cette procédure de *Branch-and-Price*. Les colonnes sont, respectivement, le nom de l'instance, la valeur optimale pour cette instance, le nombre de nœuds et le temps (en secondes) nécessaires pour prouver l'optimalité. Une stratégie de recherche de type *depth-first search* a été utilisée et les branchements sont effectués comme décrit dans la section précédente. Ces résultats ne sont toujours pas compétitifs avec ceux trouvés en utilisant d'autres méthodes.

Instance	OPT	Nb Nœuds	Temps
A04	22267	215	1,97
A05	30744	1871	13,53
A06	37716	19503	327,87

TAB. 7.2 – Résultats du *Branch-and-Price* sur la série *A*.

Le résultat le plus étonnant est la très mauvaise qualité des bornes inférieures au nœud racine, qui sont bien inférieures à la solution optimale. La table 7.3 donne une idée de l'écart entre ces bornes inférieures et les solutions optimales connues pour des instances plus grosses. La colonne *LB* indique la borne que nous obtenons et la colonne *écart* donne l'écart  $(OPT - LB)/OPT$ .

Dans d'autres situations similaires où la génération de colonnes ne donne pas de bonnes bornes inférieures, des coupes peuvent être utilisées. Comme nous l'avons vu, ceci a été proposé pour les problèmes de VRP. Nous utilisons alors une procédure de *Branch-and-Price-and-Cut* où des colonnes et des coupes peuvent être générées et ajoutées à chaque nœud.

Instance	LB	OPT/UB	écart
A04	11622,49	22267	47,80 %
A05	17098,94	30744	44,38 %
A06	17793,17	37716	52,82 %
A07	24021,86	47728	49,67 %
A08	28182,29	56576	50,19 %
A09	36082,82	70885	49,86 %
A10	42094,65	82306	48,86 %

TAB. 7.3 – Ecart entre bornes inférieures et solutions optimales pour la série A.

## 7.3 Amélioration des bornes inférieures

Les résultats des tables 7.2 et 7.3 ont montré que l'utilisation d'une procédure de *Branch-and-Price* n'a pas été suffisante pour fermer l'écart et obtenir rapidement des solutions de qualité acceptable. En particulier, nous avons souligné la mauvaise qualité de nos bornes inférieures par rapport aux solutions optimales.

Nous proposons alors d'ajouter des coupes à notre modèle afin d'améliorer ces bornes inférieures et pouvoir éliminer certaines branches de l'arbre de recherche. Ceci pourrait également nous permettre d'obtenir une relaxation plus proche de la réalité et donnant donc une meilleure information au moment de choisir les branchements à réaliser. Nous allons générer des coupes à tous les nœuds de la recherche et utiliser une méthode de *Branch-and-Price-and-Cut*.

L'utilisation de coupes dans les problèmes de conception de réseau a déjà fait l'objet de diverses études. Les coupes que nous utilisons ont presque toutes été utilisées sous une forme très proche dans l'une de ces études. Cependant, nous ne connaissons qu'un seul cas où des coupes ont été appliquées sur un modèle décomposé ([CG95]). De plus, ces études travaillant sur le modèle d'origine et n'utilisant pas de méthode de génération de colonnes, elles se limitent donc au cas plus simple du *Branch-and-Cut*.

### 7.3.1 Description des coupes utilisées

Nous avons identifié et implanté 4 types de coupes différentes :

- les coupes de type *Set* utilisent une borne inférieure sur la capacité totale des liens ouverts qui traversent la frontière d'un ensemble de nœuds donné,
- les coupes de type *BigDemand* utilisent une borne inférieure sur le nombre de liens ouverts avec une capacité supérieure à une valeur donnée qui traversent la frontière d'un ensemble de nœuds donné,
- les coupes de type *Connex* utilisent une borne inférieure sur le nombre de

liens ouverts qui ont au moins une extrémité à l'intérieur d'un ensemble de nœuds donné et respectant une propriété de connexité,

- les coupes de type *Link* contraignent les valeurs des liens pour certains arcs à être supérieures à la demande passant par ce chemin.

Ces coupes peuvent être divisées en deux catégories. La première, qui inclut les trois premiers types de coupes, ne fait intervenir que des colonnes de type *Link*. Seules ces colonnes ont des coefficients non nuls pour ces coupes. Ceci est particulièrement intéressant car ces colonnes ne sont pas générées dynamiquement mais toutes ajoutées dès le début. Ainsi, aucun modèle de sous-problème n'a besoin d'être modifié. L'autre catégorie, qui correspond au dernier type de coupes, fait intervenir les colonnes de type *Path* et doit donc être prise en compte dans les estimations de coût réduits ultérieures. Les sous-problèmes doivent être modifiés.

Nous décrivons maintenant plus complètement chacune de ces coupes.

Dans cette section, nous utilisons les notations suivantes :

- $S$  est un sous ensemble de l'ensemble  $X$  de tous les nœuds,
- $\bar{S}$  est utilisé pour  $X \setminus S$ ,
- $\delta(S)$  est utilisé pour représenter la frontière de  $S$ , i.e. le sous ensemble de  $X^2$  qui va de  $S$  à  $\bar{S}$ .

### 7.3.2 Coupes de type *Set*

Cette coupe consiste simplement à utiliser une borne inférieure sur la capacité totale nécessaire pour couvrir l'ensemble des demandes entre les nœuds de l'intérieur et de l'extérieur d'un ensemble.

Nous définissons  $demOut(S)$  la demande sortant de  $S$  comme la somme des demandes entre des nœuds intérieurs à  $S$  et des nœuds extérieurs à  $S$ .

$$demOut(S) = \sum_{p \in S} \sum_{q \notin S} Dem_{pq} = \sum_{(p,q) \in \delta(S)} Dem_{pq}$$

Nous définissons alors  $\overline{dem}(S)$  la combinaison minimale des capacités utilisables sur la frontière de  $S$  supérieure ou égale à  $demOut(S)$ . Celle-ci peut être facilement calculée, en particulier lorsque les multiplicateurs ne sont pas autorisés, le nombre total de liens utilisables étant alors limité au nombre d'arcs dans la frontière. Dans le cas général, il s'agit d'un problème de sac-à-dos. Les contraintes additionnelles peuvent également être prises en compte pour simplifier ce sous-problème consistant à calculer cette capacité couvrante minimale. Par exemple, quand la contrainte additionnelle *pmax* est utilisée, le nombre de liens sortant d'un nœud est encore plus limité, la borne utilisée peut alors être améliorée.

Nous définissons également  $cf(S)$  le flux de capacité le long des liens comme étant la somme des capacités des colonnes de type *Link* pondérées par la valeur

des capacités correspondantes, sur l'ensemble de la frontière.

$$cf(S) = \sum_{(i,j) \in \delta(S)} \sum_{k=1}^{K_{ij}} \text{Capa}^k \text{Link}_{ij}^k$$

Nous avons alors la coupe :

$$\begin{aligned} cf(S) &\geq \overline{dem}(S) \\ \sum_{(i,j) \in \delta(S)} \sum_{k=1}^{K_{ij}} \text{Capa}^k \text{Link}_{ij}^k &\geq \overline{dem}(S) \end{aligned} \quad (7.14)$$

### Cas particulier avec *CapaDiv*

Dans le cas où, comme nous l'avons commenté auparavant, les capacités sont toutes identiques et multiples d'un dénominateur commun *CapaDiv*, nous pouvons améliorer la coupe en utilisant simplement le nombre minimal de fois que la quantité *CapaDiv* est nécessaire pour couvrir le flux de demande. Simplement, cela consiste à diviser l'ensemble des résultats par *CapaDiv* et à simplifier.

Nous avons  $\overline{dem}(S)$  maintenant directement définie par :

$$\overline{dem}(S) = \text{CapaDiv} \left\lceil \frac{\text{demOut}(S)}{\text{CapaDiv}} \right\rceil$$

Nous définissons ensuite  $llf(S)$  le flux de niveau de capacité comme la somme des valeurs des colonnes de type *Link* sur la frontière de  $S$ .

$$llf(S) = \sum_{(i,j) \in \delta(S)} \sum_{k=1}^{K_{ij}} k \cdot \text{Link}_{ij}^k$$

La coupe devient alors :

$$\begin{aligned} llf(S) &\geq \overline{dem}(S) / \text{CapaDiv} \\ \sum_{(i,j) \in \delta(S)} \sum_{k=1}^{K_{ij}} k \cdot \text{Link}_{ij}^k &\geq \left\lceil \frac{\text{demOut}(S)}{\text{CapaDiv}} \right\rceil \end{aligned} \quad (7.15)$$

Cette première coupe, ainsi que sa version simplifiée, sont des cas particuliers des inégalités de *Cut-Set* introduites dans [MM93] et étudiées dans [MMR93], [MMR95], [BG96] et [Gun98].

### 7.3.3 Coupes de type *BigDemand*

Les coupes de type *BigDemand* correspondent à une situation plus spécifique où une demande, par exemple entre  $p$  et  $q$ , est supérieure à la plus petite capacité commune  $Capa^1$ . Nous définissons alors cette capacité comme étant de type *BigDemand*. Lorsqu'une demande de tel type va de l'intérieur à l'extérieur d'un ensemble de nœuds  $S$ , i.e.  $p \in S$  et  $q \in \bar{S}$ , nous pouvons définir une inégalité impliquant qu'il existe au moins un lien dans  $\delta(S)$  utilisé avec un niveau de capacité strictement supérieur à 1.

En définissant le flux de *big links* :

$$blf(S) = \sum_{(i,j) \in \delta(S)} \sum_{k>1} Link_{ij}^k$$

Nous obtenons la coupe :

$$\begin{aligned} blf(S) &\geq 1 \\ \sum_{(i,j) \in \delta(S)} \sum_{k>1} Link_{ij}^k &\geq 1 \end{aligned} \quad (7.16)$$

Cette coupe est un cas particulier des *DiCut* qui sont présentés dans la section 4.1 de [MMWW99]. Elles sont appliquées ici au *sous-problème* consistant à trouver un sous-réseau permettant d'acheminer les demandes de type *BigDemand*.

### 7.3.4 Coupes de type *Connex*

Les coupes de type *Connex* correspondent à une propriété simple des graphes connexes : dans un graphe connexe de  $n$  nœuds, nous avons au moins  $n - 1$  arcs. Dans le cas d'un sous-graphe connexe de  $n$  nœuds, nous devons donc avoir au moins  $n - 1$  arcs avec au moins une extrémité dans le sous-graphe. Nous appliquons cette propriété à un ensemble de nœuds  $S$  qui possède cette propriété de connexité pour les demandes. Nous savons alors que au moins  $card(S) - 1$  liens ayant au moins une extrémité dans  $S$  seront utilisés.

Nous définissons  $lc(S)$  la somme des valeurs des colonnes de type *Link* ayant au moins une extrémité dans  $S$ .

$$lc(S) = \sum_{i \in S \vee j \in S} \sum_{k=1}^K Link_{ij}^k$$

Si  $S$  est connexe vis-à-vis des demandes, nous avons alors la coupe :

$$\begin{aligned} lc(S) &\geq card(S) - 1 \\ \sum_{i \in S \vee j \in S} \sum_{k=1}^K Link_{ij}^k &\geq card(S) - 1 \end{aligned} \quad (7.17)$$

Nous pouvons augmenter cette coupe si nous savons qu'il existe au moins une demande entre  $S$  et  $\bar{S}$ , car la borne sur le nombre de liens est alors  $card(S)$ .

$$\begin{aligned} lc(S) &\geq card(S) \\ \sum_{i \in S \vee j \in \bar{S}} \sum_{k=1}^K Link_{ij}^k &\geq card(S) \end{aligned} \quad (7.18)$$

### 7.3.5 Coupes de type *Link*

Les coupes de type *Link* sont les seules qui font intervenir les deux types de colonnes du problème. Elles affirment simplement que si une demande transite par un arc, cet arc doit avoir au moins un lien ouvert, i.e. la somme des variables de liens doit être égale à 1. Pour un arc donné  $(i, j)$ , et une variable de chemin  $x$  telle que  $(i, j) \in x$ , la coupe est alors :

$$x \leq \sum_{k=1}^{K_{ij}} Link_{ij}^k \quad (7.19)$$

Nous pourrions vouloir étendre cette coupe à la somme sur différents chemins passant à travers l'arc  $(i, j)$ , mais nous devons être prudents car la partie gauche de la coupe pourrait alors être supérieure à 1 alors que la partie droite ne peut pas. La coupe peut par contre être étendue à la somme sur l'ensemble des chemins correspondant à la même demande  $(p, q)$  passant à travers l'arc  $(i, j)$ . Cette somme est inférieure à 1.

Pour un arc donné  $(i, j)$ , et une demande  $(p, q)$ , la coupe est alors :

$$\sum_{x \in Path_{pq}, (i,j) \in x} x \leq \sum_{k=1}^{K_{ij}} Link_{ij}^k \quad (7.20)$$

Cette coupe est plus compliquée à utiliser car elle met en jeu des colonnes de type *Path*. En effet, le coût réduit doit alors être modifié pour prendre en compte les valeurs duales associées à ces coupes.

### 7.3.6 Détails de la recherche *Branch-and-Price-and-Cut*

Nous avons utilisé une recherche type *Branch-and-Price-and-Cut* où des colonnes et des coupes peuvent être générées à chaque nœud. Une exception est faite pour les coupes de type *Link* qui ne sont générées qu'au nœud racine. Les colonnes et coupes ne sont jamais retirées du problème maître. Des précautions doivent donc être prises pour choisir les coupes à ajouter.

Les générateurs doivent alors être modifiés pour prendre en compte les nouvelles coupes. Ici, ce n'est le cas que pour les colonnes de type *Path* avec les coupes de type *Link*. Les autres coupes ne font en effet intervenir que des colonnes présentes initialement dans le problème.

Nous revenons dans cette section sur ces deux aspects.

### Algorithme de séparation heuristique

Trouver de nouvelles coupes dont l'addition au problème maître restreint résulte dans un changement important de la solution relâchée est un problème équivalent à celui consistant à trouver de bonnes colonnes. Au lieu de chercher une colonne de coût réduit le plus négatif, nous cherchons une coupe de violation la plus grande. La même variété de méthodes existe alors, allant de la génération de toutes les coupes et de leur test jusqu'aux algorithmes spécifiques mettant directement en évidence la coupe la plus violée. Cette dernière possibilité correspond au cas du problème de plus court chemin où la colonne de coût réduit le plus négatif est trouvé. Entre ces deux cas extrêmes existent de nombreux compromis permettant de trouver une *bonne* coupe parmi certaines coupes candidates.

Nous n'avons pas centré notre effort sur la recherche d'un algorithme de séparation optimal. Nous avons simplement utilisé une méthode heuristique consistant à calculer les violations pour toutes les coupes sur les ensembles contenant un nombre de nœuds inférieur à une certaine limite. Cela nous a semblé raisonnable car pour une limite de 10 nœuds, nous n'avons que 1024 ensembles candidats pour les coupes. Nous ajoutons les coupes violées de plus de  $10^{-1}$  par paquet d'au maximum 5 coupes.

Nous sommes conscients que la recherche de bons algorithmes de séparation est un domaine de recherche prometteur. Dans [OW00], différentes idées sont données sur la manière de créer un bon ensemble d'ensembles candidats.

### Modification des algorithmes de génération

Seules les coupes de type *Link* résultent en une modification des coûts réduits des colonnes de type *Path*. La modification du coût réduit est assez simple. En effet, l'expression de prix associée à la coupe nous offre une information directe sur cette modification. Dans le cas d'une coupe sur  $(p, q, i, j)$ , le coefficient pour un chemin est 1 si et seulement si il correspond à la demande  $(p, q)$  et il passe par l'arc  $(i, j)$ . L'expression de prix est donc :

$$coef_{cut} = 1_{(p,q)} * 1_{(i,j)}$$

où  $1_{(p,q)}$  vaut 1 si le chemin va de  $p$  à  $q$ , et 0 sinon, et où  $1_{(i,j)}$  vaut 1 si le chemin passe par l'arc  $(i, j)$ , et 0 sinon. Alors le coût réduit est modifié en lui

ajoutant la quantité :

$$rC_{cut} = -\pi_{cut} * 1_{(p,q)} * 1_{(i,j)}$$

Cette modification du coût réduit est facilement intégrée à tous les problèmes de plus court chemin. En effet, pour les chemins n'allant pas de  $p$  à  $q$ , le coût réduit ne change pas. Pour les chemins allant de  $p$  à  $q$ , à l'arc  $(i, j)$  est ajouté le coût  $-\pi_{cut}$ .

### 7.3.7 Résultats expérimentaux

Nous avons exécuté notre algorithme de *Branch-and-Price-and-Cut* sur toutes les instances de la série A, en utilisant les limites de temps de 2 et 10 minutes et pour la configuration de contraintes additionnelles 011000. Tous les temps donnés ici sont dans les mêmes conditions qu'auparavant et incluent le processus de génération des coupes qui ne représente qu'une petite partie du temps total d'exécution.

L'arbre de recherche est ici exploré en utilisant une heuristique de type *Best First Search* où les nœuds correspondant aux meilleures bornes sont explorés en premier. Cette méthode minimise la taille de l'arbre de recherche.

Inst.	$LB$	$LB_C$	$LB_C$ écart fermé	$LB_{CL}$	$LB_{CL}$ écart fermé
A04	11622,49	21033,00	88,41%	21033,00	88,41%
A05	17098,94	25314,00	60,21%	26237,39	66,97%
A06	17793,17	30549,50	64,03%	31808,05	70,35%
A07	24021,86	36420,20	52,30%	42042,01	76,01%
A08	28182,29	45744,38	61,85%	48670,64	72,16%
A09	36082,82	54631,29	53,30%	61619,45	73,38%
A10	42094,65	62202,40	50,01%	71813,23	73,91%

TAB. 7.4 – Améliorations des bornes inférieures au nœud racine sur la série A en utilisant les coupes.

La table 7.4 montre les bornes inférieures obtenues au nœud racine en utilisant les coupes. La colonne  $LB$  donne la borne obtenue sans utiliser les coupes,  $LB_C$  la borne obtenue en utilisant les coupes de la première catégorie et  $LB_{CL}$  la borne obtenue en utilisant les deux catégories de coupes. Pour chacune des bornes obtenues en utilisant les coupes, une colonne (*écart fermé*) donne le pourcentage de réduction de l'écart obtenu.

Dans la table 7.5 sont données les valeurs des solutions  $UB_2$  et  $UB_{10}$  obtenues après respectivement 2 et 10 minutes et en générant les coupes de la première

Instance	LB2	UB2	LB10	UB10
A07	47525,50	55776	47728	47728
A08	53387,25	89791	56576	56576
A09	60540,00	85929	63755,625	85929
A10			72208,35	105302

TAB. 7.5 – Bornes après un temps limité.

catégorie à tous les nœuds et ceux de la deuxième catégorie au nœud racine seulement. Les instances plus petites (de 4 à 6 nœuds) ne sont pas données car elles sont complètement résolues en moins de 2 minutes. La qualité des solutions obtenues est meilleure.

Instance	OPT	Nœuds	Red. Nœuds	Temps	Red. Temps
A04	22267	5	97,67%	1,56	20,81%
A05	30744	185	90,11%	2,60	80,78%
A06	37716	707	96,37%	9,40	97,13%
A07	47728	6621	*	144,19	*
A08	56576	6689	*	553,84	*

TAB. 7.6 – Résultats complets du *Branch-and-Price-and-Cut* sur la série A.

Enfin, la table 7.6 donne les résultats complets pour trouver et prouver la solution optimale. Les colonnes *Red Nœuds* et *Red. Temps* montrent les réductions en terme de nombre de nœuds et de temps obtenues. Une \* signifie qu'aucune réduction n'a pu être calculée car il a été impossible de résoudre et prouver l'optimalité de l'instance sans utiliser les coupes. Ces chiffres montrent que l'amélioration de la qualité des solutions semble provenir de la réduction de l'arbre de recherche.

Maintenant, que se passerait-il si, même réduit par l'utilisation des coupes, l'arbre restait si grand que le temps imparti ne permette pas de le parcourir complètement? Les branches intéressantes peuvent ne pas être explorées et les solutions trouvées dans le temps limité peuvent finalement être de mauvaise qualité. Ceci est le cas par exemple pour A09 et A10 comme cela est visible dans la table 7.5. Dans la prochaine section, nous proposons d'utiliser une stratégie de recherche pour le problème maître permettant d'explorer d'abord les parties de l'arbre de recherche à priori les plus prometteuses.

## 7.4 Amélioration des bornes supérieures

Dans cette section, nous présentons comment nous avons modifié heuristiquement notre algorithme de *Branch-and-Price-and-Cut* en utilisant des méta-heuristiques de manière à permettre d'obtenir des solutions de bonne qualité rapidement, sans avoir besoin d'explorer l'arbre complet. En effet, les séries *B* et *C* contiennent des problèmes plus grands où une recherche de type *Best First Search* n'est plus capable de trouver ni même une seule solution réalisable dans le temps imparti. Nous avons donc utilisé diverses limitations heuristiques à notre algorithme de *Branch-and-Price-and-Cut* afin d'explorer autant de nœuds promoteurs que possible dans la limite du temps disponible.

### 7.4.1 Utilisation de la PLNE à certains nœuds

Avec les nombreuses coupes ajoutées au problème, le nombre de variables non entières peut augmenter significativement. Il est alors plus difficile qu'une solution entière apparaisse spontanément. Nous effectuons alors une recherche normale de type PLNE à certains nœuds en utilisant uniquement certaines variables parmi celles générées jusqu'alors. Les recherches PLNE sont effectuées en utilisant ILOG CPLEX (voir [ILO02a]). Nous avons utilisé les paramètres par défaut ; seul le paramètre de *MIPemphasis* a été mis de manière à privilégier la faisabilité. Quand une solution primale est disponible, sa valeur est donnée à CPLEX pour couper l'arbre de recherche.

Deux types de recherches PLNE, *locale* et *globale*, ont été testées. Elles correspondent aux deux *goals* définis dans la section 4.2.1.

D'une part, la recherche *locale* est restreinte à l'état courant d'un nœud. Toutes les colonnes et coupes correspondant à ce nœud sont utilisées. Le but de cette recherche est de permettre l'utilisation de techniques implantées dans CPLEX telles que les heuristiques, de nombreuses coupes, etc... Certaines de ces techniques ne sont possibles que sur des problèmes dont toutes les colonnes sont connues. Même si les recherches sont limitées en nombre de nœuds et en nombre de solutions nouvelles trouvées, elle sont toujours coûteuses et ne sont donc effectuées qu'à certains nœuds sélectionnés. Notre critère a été de n'utiliser que les nœuds où le nombre de variables fractionnelles est inférieur à un niveau donné. La plupart de nos solutions sont obtenues par ce type de recherche. Il est intéressant de noter que le fait d'exécuter une recherche PLNE de ce type est similaire à ce qui serait fait (mais seulement au nœud racine) dans une procédure de génération de colonnes *simple*. Nous tendons à penser que les divers branchements effectués le long de la branche favorisent l'apparition de bonnes colonnes se complétant correctement pour donner des solutions lors de la recherche PLNE.

D'autre part, une recherche *globale* impliquant toutes les colonnes générées jus-

qu'à un point est effectuée. Les coupes et règles de branchement sont évidemment éliminées, certaines d'entre elles pouvant se contredire. L'objectif de cette recherche est légèrement différent et consiste à essayer de trouver des solutions entières qui auraient correspondu à des nœuds non visités dans le temps imparti en raison des stratégies de recherche. Ce type de recherche globale n'a pas donné de résultats probants.

### 7.4.2 Utilisation de stratégies de recherche

Nous avons vu que même avec des bornes inférieures de bonne qualité qui éliminent une grande partie de l'arbre de recherche, les stratégies habituelles d'exploration de l'arbre de recherche peuvent s'avérer incapables d'atteindre une bonne solution (et même parfois simplement d'atteindre une solution) dans le temps imparti. En effet, d'une part, une stratégie type *Depth First Search* peut plonger dans une branche et y rester durant tout le temps permis sans qu'aucune solution n'y soit présente. D'autre part, une stratégie type *Best First Search* peut dédier tout le temps à améliorer la borne inférieure globale en travaillant sur les nœuds de meilleure borne sans pour autant obtenir la moindre solution faisable. De plus, aucune recherche PLNE ne serait exécutée car les nœuds avec les meilleures bornes sont typiquement ceux avec le plus de variables fractionnelles. Nous proposons donc d'utiliser d'autres stratégies d'exploration de l'arbre de recherche.

Notre objectif, en utilisant une autre stratégie pour l'évaluation des nœuds à traiter, est de diversifier la recherche en permettant d'explorer des zones prometteuses variées. Le seul critère disponible indiquant le *potentiel* correspondant à un nœud est sa correspondance avec les décisions que nous aurions préférées. En effet, comme nous l'avons mentionné auparavant, la relaxation nous offre une image utilisée afin de guider la recherche. De plus, en utilisant des coupes, nous affinons la relaxation et rendons plus probable le fait que l'information extraite de la solution relâchée mène à des solutions entières. En résumé, nous pensons que les décisions prises en fonction de la relaxation contenant des coupes sont bonnes et que nous devrions nous baser sur cette heuristique pour estimer la *potentiel* correspondant aux nœuds.

Nous avons donc utilisé des stratégies de recherche de la famille du LDS (*Limited Discrepancy Search*) (voir [GH95]) qui permettent d'explorer d'abord les nœuds impliquant le moins de différences avec l'heuristique. Nous avons déjà utilisé cette stratégie pour résoudre le sous-problème dans le chapitre 6. Les différences mesurées sont le nombre de mouvements à droite dans l'arbre de recherche, i.e. correspondant à une décision non préférée. Cette stratégie consiste donc à donner la priorité aux branches où la stratégie de branchement originale a été la plus suivie. Notre implantation utilise plus précisément une stratégie de

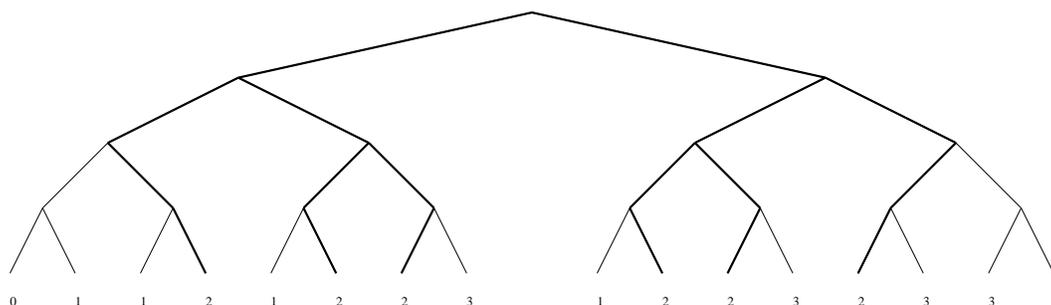


FIG. 7.1 – 2 différences.

type *Improved LDS* (ILDS) qui n'est qu'une légère optimisation du LDS original (voir [Kor96, Wal97, MW98]). D'abord, les feuilles possédant moins de différences ne sont visitées qu'une seule fois. De plus, l'ordre de visite des nœuds ayant un nombre de différences identique est légèrement différent. Une idée de l'ordre d'exploration de l'arbre de recherche est donné dans la figure 7.1 où les feuilles de nombre de différences égal à 2 sont soulignées. Dans les articles référencés sont données de plus amples informations sur les stratégies de recherche basées sur le nombre de différences.

### 7.4.3 Utilisation d'heuristique vers l'intégralité

Lorsque l'ensemble de l'arbre de recherche était parcouru, nous essayions d'utiliser les règles de branchement donnant deux sous-problèmes avec les meilleures bornes possibles<sup>1</sup>. Ceci, ajouté à une stratégie de recherche de type *Best First Search* permet de réduire la taille de l'arbre de recherche au maximum. Maintenant que nous n'explorons plus l'arbre de recherche complet mais que nous essayons de plonger rapidement vers des solutions entières, ce critère n'est plus valable pour choisir les règles de branchement à utiliser. Nous essayons alors de brancher d'une manière telle que les solutions entières puissent apparaître rapidement. Une conséquence simple de cette décision est de ne pas prendre en compte le coût des colonnes sur lesquelles les règles doivent être appliquées. Elles sont alors choisies en utilisant le critère  $|x - \text{round}(x)|$ .

### 7.4.4 Elimination de nœuds

Nous avons également utilisé des critères spécifiques afin d'éliminer des nœuds qui ne semblent pas prometteurs. Pour un nœud dont la borne inférieure locale est

<sup>1</sup>En PLNE, la méthode dite de *Strong Branching* essaie de maximiser l'amélioration de ces bornes.

$llb$  et le nombre de variables fractionnelles  $n$ , et avec la borne supérieure globale  $gub$ , nous éliminons le nœud si :

$$llb + n * variable\_dual\_bound\_step > gub$$

où  $variable\_dual\_bound\_step$  est un paramètre choisi arbitrairement. Dans nos expérimentations, il vaut 50.

## 7.5 Résultats expérimentaux

Dans cette section, nous donnons des résultats obtenus avec notre algorithme final et les comparons avec des résultats obtenus en utilisant d'autres méthodes. Comme pour les autres applications, ces résultats ont été obtenus sur notre PC portable et avec un microprocesseur à 1,1 GHz et en utilisant notre environnement de génération de colonnes et de coupes *Maestro* (voir l'annexe A). Nos résultats sont comparés avec les derniers résultats publiés (dans [PPRS02]) et avec des résultats plus récents non publiés. Comme notre microprocesseur est presque deux fois plus rapide, nous avons divisé le temps alloué par 2, i.e. nous avons utilisé une limite de temps de 5 minutes.

### 7.5.1 Comparaison des bornes supérieures

Même si initialement l'objectif du projet ROCOCO était d'obtenir à la fois de bonnes bornes supérieures et inférieures, les participants au projet ont concentré leurs efforts sur la recherche de bonnes bornes supérieures. En particulier, beaucoup de travail a été réalisé pour obtenir de bonnes solutions dans la limite des 10 minutes en utilisant la Programmation par Contraintes, la Recherche Locale, et des combinaisons de ces deux méthodes. Nous centrons nos comparaisons sur les séries B et C qui sont les seules fournissant des instances suffisamment grandes (plus de 10 nœuds). Les résultats sont d'abord comparés avec d'autres algorithmes sur les 6 instances où des résultats ont été publiés. Chaque comparaison s'effectue sur le résultat total des 64 configurations de contraintes additionnelles.

La table 7.7 effectue une comparaison avec les derniers résultats publiés dans [PPRS02]. La colonne *BPC* contient nos résultats. La colonne *CP* correspond aux résultats obtenus avec une méthode hybride entre la Programmation par Contraintes et la Recherche Locale et la colonne *Best* correspond aux meilleurs résultats parmi les autres résultats publiés (et ne correspondant donc pas toujours au même algorithme). Notre algorithme obtient de meilleurs résultats que ceux publiés pour l'algorithme *CP* et n'est battu que pour C10 par une version parallèle de l'algorithme *CP* présentée dans [PPRS02]. Il est intéressant de noter que notre méthode pourrait aussi être implantée de manière parallèle.

Instance	BPC	CP	BPC/CP	Best[PPRS02]	BPC/Best
B10	1492619	1626006	-8,20%	1592778	-6,29%
B11	2708049	3080608	-12,09%	2967717	-8,75%
B12	2320452	2571936	-9,78%	2535516	-8,48%
C10	1098593	1110966	-1,11%	1084577	1,29%
C11	1922720	2008833	-4,29%	2003101	-4,01%
C12	2443932	2825499	-13,50%	2777129	-12,00%

TAB. 7.7 – Comparaisons avec les résultats donnés dans [PPRS02]

Instance	BPC	CP	BPC/CP	Best	BPC/Best
B10	1488661	1610770	-7,58%	1559876	-4,57%
B11	2683531	3023086	-11,23%	2867269	-6,41%
B12	2308483	2555469	-9,66%	2495065	-7,48%
B15	2240244	2616749	-14,39%	2545590	-12,00%
B16	2309586	2521154	-8,39%	2482189	-6,95%
B20	4038697	4809675	-16,03%	4633576	-12,84%
B25	6386837	6878867	-7,15%	6792567	-5,97%
C10	1098593	1110966	-1,11%	1074942	2,20%
C11	1922720	2003101	-4,01%	1856626	3,56%
C12	2443932	2801849	-12,77%	2603355	-6,12%
C15	3635519	4207669	-13,60%	3860496	-5,83%
C16	1768982	2013729	-12,15%	1858298	-4,81%
C20		7218196		6990726	
C25	6999698	7444034	-5,97%	7340627	-4,64%

TAB. 7.8 – Comparaisons des bornes supérieures

Dans la table 7.8, nos résultats sur de plus nombreuses instances sont comparés aux derniers résultats non publiés pour l'algorithme hybride précédent. Notre algorithme fournit toujours de meilleurs résultats et est battu par une approche basée sur le LNS (*Large Neighbor Search*) sur 2 instances. Comme nous l'avons signalé auparavant, nous ne donnons pas de résultats pour C20 qui contient des demandes multiples pour une même paire origine-destination.

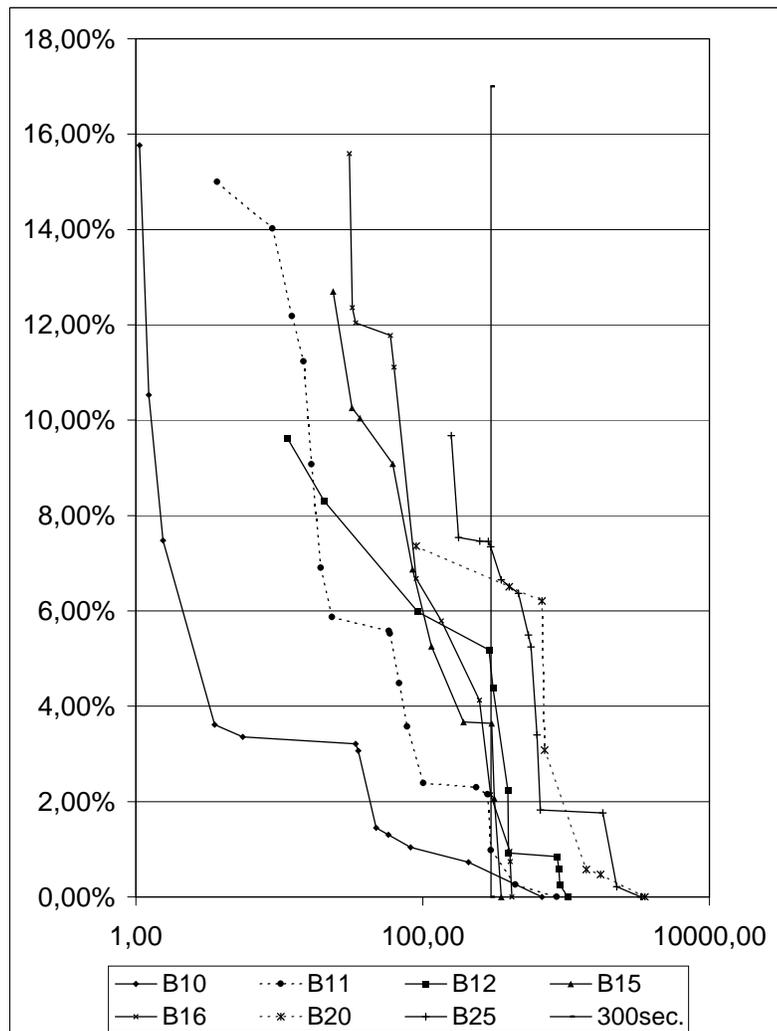


FIG. 7.2 – Evolution des solutions trouvées

### 7.5.2 Influence de la limite de temps et des contraintes additionnelles

Nous voulions également savoir si la limite de temps impartie était réellement restrictive. Pour cela, nous avons exécuté notre algorithme sur la configuration 111111 (avec toutes les contraintes additionnelles) pour toutes les instances de la série B en étendant la limite de temps à 1 heure. Le graphique 7.2 montre l'évolution des solutions trouvées au cours du temps. Pour une meilleure lisibilité, l'axe horizontal représentant le temps utilise une échelle logarithmique. Nous pouvons observer que l'objectif évolue par paliers, chacun pouvant correspondre à plusieurs solutions nouvelles. Nous attribuons ce phénomène à la stratégie de recherche permettant de visiter une nouvelle zone de l'arbre de recherche. Après 300 secondes, seul *B10* a atteint une solution à moins de 2 % de la solution atteinte en 1 heure. Toutes les autres instances se verront donc fortement affectées par la limite. Après 10 minutes, seuls *B20* et *B26* n'ont toujours pas atteint de solution à moins de 2 % de leur solution finale. Ce graphique nous suggère qu'une amélioration d'un facteur 2 de notre implantation, sans changer la complexité théorique de la méthode, pourrait permettre une nouvelle amélioration des solutions.

## 7.6 Conclusions

Dans ce chapitre, une nouvelle application des méthodes de génération de colonnes avec sous-problèmes de plus court chemin a été présentée. Cependant, les résultats obtenus en utilisant une procédure simple se sont avérés très mauvais. Nous avons alors proposé plusieurs améliorations portant sur le problème maître pour améliorer la qualité des résultats.

Tout d'abord nous avons incorporé des coupes à notre algorithme. Ces coupes correspondent à des idées simples sur la réalité du problème et sont possibles en génération de colonnes grâce à la connaissance des structures internes des colonnes. Ces coupes ont permis d'améliorer significativement les bornes obtenues. De telles coupes seront applicables dans la majorité des modèles de génération de colonnes avec sous-problèmes de plus court chemin.

Aussi, nous avons utilisé une stratégie de recherche permettant de visiter l'arbre de recherche du problème maître dans un ordre favorisant des parties que nous considérons prometteuses.

Enfin, nous avons appliqué quelques modifications heuristiques au problème maître. L'élimination directe de nœuds peu prometteurs et la recherche PLNE aux nœuds prometteurs nous ont permis d'obtenir rapidement des solutions de qualité.

Les résultats expérimentaux ont permis de vérifier que ces deux dernières réductions heuristiques permettent d'obtenir des solutions de qualité dans un temps réduit. Ces même réductions seront applicables à l'ensemble des méthodes de génération de colonnes.

# Conclusion

Dans cette thèse, nous avons présenté des généralisations et des améliorations des méthodes de génération de colonnes et de coupes appliquées aux cas où les sous-problèmes sont des problèmes de plus court chemin.

Nous avons tout d'abord réalisé un inventaire des différentes modifications de la méthode d'origine qui ont été publiées. Alors que celles-ci permettent parfois de réduire significativement les temps de résolution, elles n'ont souvent été utilisées que pour certains types particuliers d'applications de la génération de colonnes.

Nous avons ensuite proposé des formalismes génériques permettant une modélisation des problèmes traitables par génération de colonnes ainsi qu'une description des méthodes de résolution. Ces formalismes permettent alors une définition générique des méthodes d'accélération qui peuvent ensuite être appliquées plus facilement à des problèmes différents.

Deux bibliothèques C++ reprenant ces idées ont également été implantées. Elles offrent une interface de programmation facilitant la mise en œuvre des méthodes décrites dans cette thèse.

Ensuite, nous avons proposé diverses modifications originales de la méthode. Parmi celles-ci, citons :

- plusieurs modifications de l'algorithme de plus court chemin habituellement utilisé permettant de résoudre le problème sans cycle,
- l'utilisation d'heuristiques et de stratégies de recherche pour résoudre le sous-problème et le problème maître,
- une contrainte globale de plus court chemin utilisant un algorithme de filtrage correspondant au problème de plus court chemin avec fenêtres de temps et contraintes de ressources,
- l'utilisation de coupes sur un modèle décomposé.

Pour valider nos propositions, nous avons présenté des résultats expérimentaux obtenus avec l'environnement de génération de colonnes que nous avons développé et mettant en œuvre nos propositions. Trois familles d'applications ont été abordées : la tournée de véhicules, la planification de ressources, et la conception de réseau. Pour le problème de tournée de véhicules, nous avons fermé certaines instances reconnues en démontrant l'optimalité de nos solutions. Pour le problème de conception de réseau, nous obtenons de nouveaux meilleurs résultats sur un

benchmark public. Enfin, pour le problème de génération de pairings où aucun benchmark n'existe, nous avons pu comparer avantageusement nos résultats avec ceux obtenus manuellement par un expert.

Les propositions faites dans cette thèse sont particulièrement orientées sur la possibilité de généraliser certaines améliorations pour différents types de problèmes. Ceci semble particulièrement prometteur vu le nombre grandissant d'applications envisagées. A celles présentées dans cette thèse, nous pourrions en ajouter de nombreuses autres. Simplement dans l'industrie du transport aérien, citons le *Fleet Assignment Problem* ou l'*Aircraft Rotation Problem*. Le premier problème possède une structure similaire au problème de conception de réseau et le deuxième au problème de tournée de véhicules. De façon plus générale, tous les problèmes assimilables à des problèmes de flux dans un graphe avec une (ou plusieurs) commodité(s) et une (ou plusieurs) facilité(s) peuvent profiter avantageusement des méthodes proposées. Enfin, au delà des situations où le sous-problème apparaît directement sous une forme de problème de plus court chemin, citons deux familles de problèmes où nous pensons que la génération de colonnes pourrait avoir un impact prochainement : le *Combinatorial Auction Problem* et le *Portfolio Optimization*. Même si la structure du sous-problème y est différente, nos formalismes peuvent être utilisés et ainsi des méthodes d'accélération déjà développées leur être facilement appliquées.

Les méthodes de génération de colonnes sont souvent considérées marginales. Nous voyons deux raisons à cela. La première est la complexité apparente de la méthode. Nous avons déjà signalé plusieurs fois qu'il n'est finalement pas nécessaire de comprendre l'ensemble de la méthode de décomposition pour mettre en œuvre les méthodes de génération de colonnes. Cependant, il reste que l'existence de deux problèmes de natures très variées, utilisant des notations et notions différentes, ainsi que la définition des interactions entre eux, rebute certains. Nous espérons que la formalisation proposée ici, associée à l'utilisation de notre implantation, permette une prochaine amélioration de cette situation. La deuxième raison est l'association systématique de la génération de colonnes aux problèmes linéaires. Nous avons vu que les sous-problèmes peuvent contenir des types de contraintes et des formes de coûts très variés. Il est cependant vrai que les contraintes entre colonnes, i.e. celles présentes au niveau du problème maître, doivent à priori être linéaires. Cette condition peut parfois être très restrictive.

De nombreux axes de recherche relatifs à la génération de colonnes sont actuellement ouverts. En particulier, il nous semble que le formalisme des expressions de prix doit pouvoir être étendu afin de permettre l'écriture de systèmes de contraintes représentant certaines familles de contraintes non linéaires. D'autre part, l'utilisation de la recherche locale au niveau du sous-problème ou du problème maître n'a presque pas été étudiée, alors qu'elle permettrait, à notre avis, de rapprocher les méthodes de génération de colonnes des algorithmes génétiques.

Nous travaillons actuellement sur ces deux axes de recherche ainsi qu'à l'amélioration de la facilité d'utilisation de nos bibliothèques.



# Annexe A

## Nos implantations : *Maestro* et *ShortestPath*

Un outil facilitant la mise en œuvre des méthodes de génération de colonnes et de coupes et reprenant les idées de cette thèse a été implanté. Celui-ci a été utilisé pour tous les tests, applications et résultats donnés dans cette thèse. Deux bibliothèques C++ sont disponibles : *Maestro* et *ShortestPath*, qui offrent respectivement des fonctionnalités pour résoudre des problèmes en utilisant les méthodes de *Branch-and-Price-and-Cut* et pour résoudre les problèmes de plus court chemin contraints qui apparaissent dans les sous-problèmes.

Nous présentons cet outil brièvement dans cette annexe et donnons des exemples de son utilisation. Pour plus de détails, il est possible de se reporter aux manuels d'utilisation correspondant.

Ces bibliothèques C++ ont été développées en collaboration avec Javier Lafuente.

### A.1 *Maestro*

*Maestro* est une bibliothèque permettant :

- de modéliser le problème de génération de colonnes et de coupes d'une manière orientée objets et en s'appuyant en partie sur le formalisme de modélisation présenté ici. Les notions de colonnes, coupes et règles de branchement sont en effet directement accessibles et manipulables via des objets simples.
- de faciliter l'intégration de cette méthode d'optimisation dans une application industrielle en étant peu intrusif,
- de mettre en œuvre une recherche arborescente compatible avec l'ajout de colonnes et de coupes à n'importe quel nœud. Cette recherche est facilement décrite en utilisant des *goals* et des évaluateurs.

- de paralléliser la recherche de plusieurs manières en minimisant les changements de codes,
- d'utiliser une génération complète ou partielle des colonnes et/ou des coupes,
- et enfin, d'appliquer facilement un certain nombre de méthodes accélérant la recherche.

Notons que ILOG CPLEX ([ILO02a]) est utilisé pour résoudre les problèmes linéaires apparaissant aux différents nœuds de la recherche.

### A.1.1 IloMaestro

La principale classe, `IloMaestro`, représente l'algorithme. Elle est utilisée pour :

- extraire le problème maître,
- ajouter des colonnes et solutions initiales, définir des variables d'écart,
- maintenir les structures globales et obtenir les résultats locaux et globaux.

Un exemple d'utilisation de cette classe pour définir un problème maître du problème de découpe (section 1.3.1) est :

```
IloEnv env;

IloModel model(env);
Objective = IloAdd(model, IloMinimize(env));
DemandRanges =
    IloAdd(model, IloRangeArray(env, N, demands));

IloMaestro maestro(model);
maestro.solve(goal);
```

### A.1.2 IloBPColumn

La classe `IloBPColumn` permet de définir de nouveaux types de colonnes. Seule une méthode doit être écrite et ceci peut être fait en utilisant des macros C++. Cette méthode doit construire la colonne de la matrice correspondante. Par exemple :

```
ILOBPCOLUMN1(Pattern,
             IloIntArray, n) {
    return Objective(1) + DemandRanges(n);
}
```

Une colonne peut ensuite être créée et utilisée avec :

```
IloBPColumn pattern =
    Pattern(env, 0, IloInfinity, elements);
```

Dans le cas du VRP (chapitre 5), la colonne de route est donnée simplement par :

```
ILOBPCOLUMN3(Route,
              IloNum, cost,
              IloIntArray, visits,
              IloInt, vehicle) {
    IloNumColumn col = Objective(cost);
    for (int i=0; i<visits.getSize(); i++)
        col += VisitRanges[visits[i]](1);
    col += VehicleRanges[vehicle](1);
    return col;
}
```

### A.1.3 Goals

Comme nous l'avons proposé dans le chapitre 4, les goals sont utilisés pour décrire la recherche. Des goals par défaut sont proposés pour ajouter ou retirer des colonnes ou des coupes. Des macros C++ sont fournies pour définir facilement de nouveaux goals.

Un goal générant des colonnes est alors facilement écrit :

```
ILOBPGOAL1(PriceGoal, MyData *, data){
    IloNumArray duals(getEnv());
    getDuals(duals, data->getMRanges());
    IloBPColumnArray cols(getEnv());

    // Generer de nouvelles colonnes
    // et les ajouter au tableau

    if (cols.getSize())
        return IloBPAnd(cols, this);
    return NULL;
}
```

Les goals sont facilement combinés en utilisant les goals par défaut. Par exemple :

```
IloBPAnd(HeuristicPriceGoal,
         CompletePriceGoal);
```

### A.1.4 IloBPBranchingRule

Cette classe est utilisée pour définir les projections des règles de branchement sur les sous-problèmes. Seule une méthode doit être écrite définissant si la règle est violée ou non par une colonne reçue en argument. Des macros C++ sont fournies

## 198ANNEXE A. NOS IMPLANTATIONS : MAESTRO ET SHORTESTPATH

pour faciliter la définition de nouvelles règles de branchement. Par exemple, la règle de branchement sur les arcs, utilisée dans le VRP (chapitre 5) ou dans la conception de réseau (chapitre 7), est donnée par :

```
ILOBPBRANCHINGRULE3(ArcBranchingRule,
                    IloInt, i,
                    IloInt, j,
                    IloBool, sense) {
  if (column->isType(PathI::GetTypeInfo())) {
    PathI* path = (PathI*)column;
    if (sense)
      return (path->containsNode(i)
              || path->containsNode(j)
              && !path->containsArc(i, j));
    else
      return path->containsArc(i, j);
  }
  return IloFalse;
}
```

Les branchements sont alors facilement réalisés dans des goals, comme le montre le goal suivant :

```
ILOGOAL1(BranchGoal, MyData *, data) {
  IloColumnArray cols = getColumns();

  IloBPBranchingRule r1, r2;

  // Iterer sur les colonnes,
  // definir deux regles de branchement et
  // les garder dans r1 et r2.

  if (r1.getImpl())
    return IloBPAnd(IloBPOr(r1, r2),
                    PriceGoal,
                    this);
  return NULL;
}
```

Comme nous l'avons précisé, le goal de génération de colonnes doit alors être modifié en conséquence :

```
ILOBPGOAL1(PriceGoal, MyData *, data){
  IloNumArray duals(getEnv());
  getDuals(duals, data->getMRanges());
}
```

```

IloBPBranchingRuleIterator rls = getRules();
IloBPColumnArray cols(getEnv());

// Generer de nouvelles colonnes
// en tenant compte des regles
// et les ajouter au tableau

if (cols.getSize())
    return IloBPAnd(cols, this);
return NULL;
}

```

### A.1.5 IloBPCut

Cette classe permet de définir simplement des coupes. Une seule méthode doit être écrite retournant la valeur de l'expression de prix pour cette coupe et pour une colonne reçue en argument. Des macros sont fournies pour faciliter la définition de nouvelles coupes. Celles ci sont facilement ajoutées et retirées lors de la recherche en utilisant des goals pré-définis. Par exemple, la coupe de type *Set* de la conception de réseau (chapitre 7) peut s'écrire :

```

ILOBPCUT2(CutSet, Set*, set, Data*, data) {
    if (column->isType(Link::GetTypeInfo())) {
        Link* link = (Link*)column;
        if (set->crossFrontier(link->_i, link->_j))
            return data->Capacities[link->_k];
        return 0;
    }
}

```

### A.1.6 IloBPNodeEvaluator

Des évaluateurs permettent d'appliquer facilement une stratégie de recherche à l'arbre de recherche ou à une sous-partie spécifique de celui ci.

### A.1.7 Parallélisme

Deux types de parallélisme sont disponibles :

- parallélisation du traitement des nœuds. Chaque processeur traite un nœud différent pris dans une liste commune de nœuds ouverts. Le nombre de processeurs utilisés est simplement passé au constructeur de `IloMaestro` :  
`IloMaestro maestro(env, nbThreads);`

- parallélisation des actions appliquées à un même nœud. Ici les actions étant définies par des goals, celles-ci peuvent être directement exécutées en parallèle en utilisant :

```
IloBPPAnd(goal1, goal2, goal3);
```

## A.2 *ShortestPath*

La bibliothèque C++ *ShortestPath* permet de :

- résoudre les problèmes de plus court chemin avec contraintes de ressources et fenêtres de temps apparaissant dans les sous-problèmes,
- résoudre les problèmes contenant des contraintes additionnelles en s'intégrant sous forme de contrainte globale dans un environnement de programmation par contraintes ,
- s'intégrer facilement dans la recherche de *Maestro*.

Nous donnons simplement un exemple d'utilisation de cette bibliothèque.

```
#include <ilshortestpath/ilshortestpath.h>
#include <ilconcert/ilomodel.h>

// The graph is
//
// 3 ----> 1 -----> 2
//  \      ^      /
//  \      |      /
//  \      v      /
//   --> 0--/
//
// Arc 0-1 is bidirectional
// Costs are all 1 except between 0 and 1 which is -1 both sense.
// Length is limited to 5 arcs.

ILOSTLBEGIN

IloNum LengthFunction(IloInt i, IloInt j) {
    if (i==j) return 0;
    return 1;
}

IloNum CostFunction(IloInt i, IloInt j) {
    if (i==j) return 0;
    if (i==0 && j==1) return -1;
    if (i==1 && j==0) return -1;
```

```

    return 1;
}

int main(int, char**) {
    IloEnv env;
    try {
        IloModel model(env);
        IloInt nbNodes = 4;
        IloIntVarArray nexts(env, nbNodes);
        // Sparse graph definition
        nexts[0] = IloIntVar(env, IloIntArray(env, 2, 1, 2));
        nexts[1] = IloIntVar(env, IloIntArray(env, 2, 0, 2));
        nexts[2] = IloIntVar(env, IloIntArray(env, 1, 3));
        nexts[3] = IloIntVar(env, IloIntArray(env, 2, 0, 1));
        IloIntVarArray length(env, nbNodes, 0, 5);
        IloIntVarArray cost(env, nbNodes, -100, 100);
        model.add(IloPathLength(env, nexts, length, LengthFunction));
        model.add(IloPathLength(env, nexts, cost, CostFunction));
        IloExpr costExpr = cost[nbNodes-2];
        model.add(IloMinimize(env, costExpr));

        IloShortestPath sp(env, nexts);

        sp.extract(model);
        if (sp.solve()) {
            cout << "Solution" << endl;
            cout << "Cost = " << sp.getObjValue() << endl;

            for (IloPathIterator it(sp); it.ok(); ++it)
                cout << *it << " ";
            cout << endl;
        } else {
            cout << "No Solution" << endl;
        }

        sp.end();

    } catch(IloException& e) {
        cerr << e << endl;
    }
    env.end();
    return 0;
}

```

## A.3 Un exemple complet

Dans cette section, nous décrivons le code complet pour un problème. Nous avons choisi un exemple simple qui permet d'obtenir les bornes inférieures pour le problème de tournées de véhicules décrites dans le chapitre 5.

### A.3.1 Fonction principale

La fonction principale de cet exemple est :

```
// Main
int main(int argc, char** args) {
    IloEnv env;
    try {
        VRP problem("RC203", 25);

        ReadArgs(argc, args, &problem);

        //-----
        // DATA
        //-----
        ReadSolomonData(&problem);

        //-----
        // MASTER problem
        //-----
        IloMaestro maestro(env);
        MakeMaster(&problem, maestro);
        maestro.setDisplayPeriod(1);

        AddSlacks(&problem, maestro);

        IloBPGoal g = Generate(env, &problem);
        maestro.solve(g);

        cout << "Total time = " << env.getTime() << endl;

        maestro.end();
    } catch(IloException& e) {
        cerr << "Error " << e << endl;
    }
    env.end();
    return 0;
}
```

```
}

```

Cette fonction :

- crée un objet qui conserve les données du problème,
- lit et interprète les éventuels arguments passés au programme,
- lit les données en fonction de l’instance utilisée précédemment pour créer le problème,
- crée une instance de l’algorithme,
- crée le problème maître,
- ajoute des variables d’écart afin qu’une solution initiale soit disponible,
- appelle la recherche avec un goal qui crée de nouvelles tournées et se rappelle récursivement.

Nous donnons et décrivons maintenant les principaux éléments du programme. Seuls quelques parties triviales comme la lecture des données, ne sont pas reproduits ici. Cet exemple, ainsi qu’une vingtaine d’autres, sont inclus dans la distribution actuelle de *Maestro*.

### A.3.2 La classe de colonnes

Les colonnes sont des instances de la classe `Route` :

```
////////////////////////////////////
// The route column class.
//
class RouteI : public IloBPColumnI {
    ILOBPCOLUMNDECL
    VRP* _problem;
    IloNum _cost;
    int* _nexts;
public:
    RouteI(IloEnvI* env, VRP* problem, IloNum cost, int* nexts);
    virtual IloNumColumn makeColumn() const;
    virtual void display(ILOSTD(ostream)& out) const;
    IloBool contains(IloInt i) const {
        return (_nexts[i] != -1);
    }
    IloInt getNext(IloInt i) const {
        return _nexts[i];
    }
};
RouteI::RouteI(IloEnvI* env, VRP* problem, IloNum cost, int* nexts)
    : IloBPColumnI(env), _problem(problem), _cost(cost), _nexts(nexts) {}

```

```

IloNumColumn RouteI::makeColumn() const {
    IloNumColumn col = _problem->Objective(_cost);
    for (int i=0; i<_problem->NbVisits; i++) {
        if (_nexts[i] != -1)
            col += _problem->VisitRanges[i](1);
    }
    col += _problem->VehicleRange(1);
    return col;
}

void RouteI::display(ostream& out) const {
    out << _cost << " : ";
    int i = _problem->NbVisits+1;
    while (i!=_problem->NbVisits) {
        out << i << " ";
        i = _nexts[i];
    }
}

```

### A.3.3 Création du problème maître

Cette fonction crée un modèle qui contiendra le problème maître et y ajoute les contraintes maîtres. Le modèle est ensuite extrait par l'algorithme.

```

////////////////////
// Function creating the master problem.
//
void MakeMaster(VRP* problem, IloMaestro maestro) {
    //-----
    // MASTER PROBLEM
    //-----
    IloEnv env = maestro.getEnv();

    IloModel model(env);
    // Master Objective
    problem->Objective = IloAdd(model, IloMinimize(env));

    // Master-Problem Common Constraints
    problem->VisitRanges = IloRangeArray(env, problem->NbVisits);
    for (int i=0; i<problem->NbVisits; i++)
        problem->VisitRanges[i] = IloAdd(model, IloRange(env, 1, IloInfinity));
    problem->VehicleRange = IloAdd(model,
        IloRange(env, -IloInfinity, problem->Params->MaxNbVehicles));

    maestro.extract(model);
}

```

```
}

```

### A.3.4 Ajout des variables d'écart

```
////////////////////
// Function adding slacks with sa specified cost.
// If no cost is given, then MaxTime (which is an upper bound
// on route costs is used.
//
void AddSlacks(VRP* problem, IloMaestro maestro, IloNum delta) {
    const IloNum slackInitialBound = 10;
    if (delta<0)
        delta = problem->Data->MaxTime;
    for (int i=0; i<problem->NbVisits; i++) {
        maestro.makePosSlack(problem->VisitRanges[i],
                             delta, slackInitialBound);
    }
}

```

### A.3.5 Goal de recherche

C'est la partie peut être la plus complexe, car elle doit décrire et résoudre le problème de plus court chemin.

```
ILOBPGOAL1(Generate,
           VRP*, problem) {
    display();

    try {
        IloEnv localEnv;
        IloInt i;

        // Create the complete graph
        IloSPDiGraph diGraph(localEnv, problem->NbVisits+2, IloTrue);

        // Create the path variable
        IloSPPathVar pathVar(localEnv, diGraph);

        // Create the subproblem model
        IloModel model(localEnv);

        // We have 3 dimensions : time, length, and capacity.
        IloSPNumArcDimension timeDimension(diGraph,
                                           TimeTransit(localEnv, problem));
    }
}

```

206ANNEXE A. NOS IMPLANTATIONS : MAESTRO ET SHORTESTPATH

```

IloSPNumVarNodeDimension time(diGraph, 0, problem->Data->MaxTime);
model.add(IloSPPathLength(localEnv, pathVar,
                          time, timeDimension));

IloSPNumArcDimension lengthDimension(diGraph,
                                     LengthTransit(localEnv, problem));
IloSPNumVarNodeDimension length(diGraph, 0, IloInfinity);
model.add(IloSPPathLength(localEnv, pathVar,
                          length, lengthDimension));

IloSPIntArcDimension capacityDimension(diGraph,
                                       CapacityTransit(localEnv, problem));
IloSPIntVarNodeDimension capacity(diGraph, 0, problem->Data->Capacity);
model.add(IloSPPathLength(localEnv, pathVar,
                          capacity, capacityDimension));

// Another dimension is used for the dual part of cost.
IloNumArray pis(localEnv);
getDuals(pis, problem->VisitRanges);
pis.add(getDual(problem->VehicleRange));
pis.add(0);
IloSPNumArcDimension dualDimension(diGraph, DualTransit(localEnv, pis));
IloSPNumVarNodeDimension dual(diGraph, -IloInfinity, IloInfinity);
model.add(IloSPPathLength(localEnv, pathVar, dual, dualDimension));

// Create cost and reduced-cost expression
IloExpr costExpr = problem->Params->constCost
                + length.getVar(problem->NbVisits);
IloExpr reducedCostExpr = costExpr - dual.getVar(problem->NbVisits);
// Minimize reduced-cost
model.add(IloMinimize(localEnv, reducedCostExpr));
// Constraint reduced-cost to be negative
model.add(reducedCostExpr <= -problem->Params->Epsilon);

// Add time windows
for (i=0; i<problem->NbVisits; i++)
    model.add( problem->Data->TimeMin[i]
              <= time.getVar(i) <=
              problem->Data->TimeMax[i] );

// Use Shortest Path to solve this subproblem.
IloShortestPath sp(localEnv, pathVar);
// Reduce heuristically the problem.
sp.setMinNbPaths(20, 10000);

```

```

// Extract the model
sp.extract(model);

// Search for routes and keep them in an array
IloBPColumnArray routes(getEnv());
sp.newSearch();
while (sp.next()) {
    IloNum cost = problem->Params->constCost
                + sp.getValue(length.getVar(problem->NbVisits));
    int* visits = new (getEnv()) int[problem->NbVisits+2];
    for (i=0; i<problem->NbVisits+2; i++)
        visits[i] = (sp.getValue(pathVar.getNextVar(i))!=i)
                    ? (IloInt)sp.getValue(pathVar.getNextVar(i))
                    : -1;
    routes.add(Route(getEnv(), problem, cost, visits));
    if (routes.getSize()>=20) break;
}
sp.endSearch();
sp.end();
localEnv.end();

if (routes.getSize())
    return IloBPAnd(routes, this);
} catch (IloException& e) { cerr << e << endl; }

return NULL;
}

```

Les dimensions utilisées sont créées de la manière suivante :

```

////////////////////
// This is the length transit
// Basically calculates the distance between i and j.
// Depending on RoundDistances param, distances are
// rounded to 0.1 or not.
//
ILOSPPATHTRANSIT1(LengthTransit,
                  VRP*, problem) {
    if (i==j) return 0;
    int ax, ay, bx, by;
    if ( (i==problem->NbVisits+1) && (j<problem->NbVisits) ) {
        ax = problem->Data->DepotX;
        ay = problem->Data->DepotY;
        bx = problem->Data->PosX[j];
        by = problem->Data->PosY[j];
    }
}

```

```

}
else if ( (i<problem->NbVisits) && (j<problem->NbVisits) ) {
    ax = problem->Data->PosX[i];
    ay = problem->Data->PosY[i];
    bx = problem->Data->PosX[j];
    by = problem->Data->PosY[j];
} else if ( (i<problem->NbVisits) && (j==problem->NbVisits) ) {
    ax = problem->Data->PosX[i];
    ay = problem->Data->PosY[i];
    bx = problem->Data->DepotX;
    by = problem->Data->DepotY;
}
else return 0;
if (problem->Params->RoundDistances)
    return ((double)IloFloor(10*(sqrt( (ax-bx)*(ax-bx)
                                     + (ay-by)*(ay-by) ))))/10.;
else
    return sqrt( (ax-bx)*(ax-bx) + (ay-by)*(ay-by) );
}

```

## A.4 Comparaisons avec d'autres systèmes

D'autres environnements de génération de colonnes et de coupes ont été développés. Citons par exemple :

- ABACUS ([JT97a, JT97b, JTb, JTa, Böh99]),
- MINTO ([SNS94, SN95, SN98]),
- Symphony-COIN/BCP ([RLc, RLb, RLa, RLTJ]).

Notre système reprend les idées sur la généralisation de la génération de colonnes développées dans cette thèse. Aussi, l'expérience en développement acquise au sein de l'équipe de Recherche et Développement de ILOG est mise à profit pour obtenir une interface de programmation très simple.

**Orientation objet.** A une colonne, coupe ou règle de branchement correspond un objet C++. Ceci rend la définition de nouveaux types et leur manipulation très facile. L'apprentissage du système est réduit au minimum, grâce en particulier à l'utilisation de Concert Technology pour la définition du problème maître et du sous-problème utilisant Concert Technology et à l'utilisation de macros C++ pour décrire les nouvelles colonnes, coupes et règles de branchement.

**Description de la recherche par des goals.** Dans les autres systèmes, la procédure de recherche suit un organigramme précis et pré-défini. Les goals per-

mettent de re-définir complètement la recherche. Les heuristiques et stratégies de recherche pouvant être définies sont illimitées.

**Disponibilité d'algorithmes pour le sous-problème.** Le sous-problème défini en utilisant Concert Technology peut être directement résolu en utilisant l'un des algorithmes existants de PPC (ILOG Solver), RL (ILOG Solver LS) ou PL (ILOG CPLEX).

**Algorithme de plus court chemin.** En particulier si le sous-problème est un problème de plus court chemin, une contrainte globale pour ILOG Solver incorporant un algorithme de programmation dynamique efficace est disponible.

**Méthodes d'accélération pré-définies.** De même que dans les autres systèmes, certaines des méthodes d'accélération décrites dans cette thèse sont intégrées au traitement du problème maître et fonctionnent de manière transparente pour l'utilisateur.

**Facile parallélisation.** Deux types de parallélisation sont disponibles : la parallélisation des traitements des nœuds de l'arbre de recherche (comme en PLNE) et la parallélisation de différents générateurs à un même nœud de la recherche. Chacune des parallélisations peut être obtenue en modifiant une ligne du programme.

210ANNEXE A. NOS IMPLANTATIONS : MAESTRO ET SHORTESTPATH

# Table des figures

1.1	Structure centrale des algorithmes de génération de colonnes . . . .	27
1.2	Exemple illustratif . . . . .	28
1.3	Exemple de patron de découpe. . . . .	31
2.1	Solution de l'exemple de VRP . . . . .	52
2.2	Illustration de la possible interaction entre génération heuristique et génération complète . . . . .	69
2.3	Réduction du graphe utilisant le coût réduit . . . . .	74
5.1	LB(1) pour RC203.25 . . . . .	132
5.2	LB pour RC203.25 . . . . .	133
6.1	Exemple de vol, activité aérienne et pairing . . . . .	139
6.2	Le vol de situation n'est pas nécessaire. . . . .	142
6.3	Le vol de situation est nécessaire. . . . .	142
6.4	Limitation s'appliquant sur chaque intervalle . . . . .	148
6.5	Grappe illustrant le fonctionnement de la contrainte globale . . . .	152
6.6	Exemple d'arbre de recherche pour un problème pur. . . . .	153
6.7	Exemple d'arbre de recherche pour un problème non pur. . . . .	153
6.8	LDS avec largeur max. 1 . . . . .	156
6.9	LDS avec largeur max. 2 . . . . .	156
6.10	LDS avec largeur max. 3 . . . . .	157
6.11	Evolutions des résultats dans le temps . . . . .	161
7.1	2 différences. . . . .	185
7.2	Evolution des solutions trouvées . . . . .	188



# Liste des tableaux

2.1	Exemple de distancier . . . . .	51
2.2	Équivalences <i>Branch-and-Price</i> et <i>Branch-and-Cut</i> . . . . .	53
2.3	Définition du SPRCTW . . . . .	60
4.1	Correspondances entre applications et propositions . . . . .	113
5.1	Bornes inférieures sur la série 1 . . . . .	129
5.2	Bornes inférieures sur la série 2 . . . . .	130
5.3	Les instances que nous avons fermées . . . . .	134
6.1	Types d'éléments collectables . . . . .	150
6.2	Consommations de ressource . . . . .	151
6.3	Résultats sur Crew . . . . .	159
6.4	Meilleurs résultats obtenus. . . . .	159
7.1	Résultats de la génération de colonnes simple sur la série A. . . . .	172
7.2	Résultats du <i>Branch-and-Price</i> sur la série A. . . . .	174
7.3	Écarts entre bornes inférieures et solutions optimales pour la série A. . . . .	175
7.4	Améliorations des bornes inférieures au nœud racine sur la série A en utilisant les coupes. . . . .	181
7.5	Bornes après un temps limité. . . . .	182
7.6	Résultats complets du <i>Branch-and-Price-and-Cut</i> sur la série A. . . . .	182
7.7	Comparaisons avec les résultats donnés dans [PPRS02] . . . . .	187
7.8	Comparaisons des bornes supérieures . . . . .	187



# Bibliographie

- [AFP98] R. Anbil, J.J. Forrest, and W.R. Pulleyblank, *Column generation and the airline crew pairing problem*, vol. III, pp. 677–686, Documenta Mathematica - Extra Volume ICM, 1998.
- [BCP<sup>+</sup>02] R. Bernard, J. Chambon, C. Le Pape, L. Perron, and J.C. Régin, *Résolution d'un problème de conception de réseau avec parallel solver*, Journées Françaises de Programmation Logique par Contraintes, 2002.
- [Bel57] R. Bellman, *Dynamic programming*, Princeton, NJ, USA, 1957.
- [BG96] D Bienstock and O. Gunluk, *Capacitated network design - polyhedral structure and computation*, INFORMS Journal on Computing **8** (1996), no. 3, 243–259.
- [Böh99] Max Böhm, *Parallel abacus - introduction and tutorial*, Tech. Report ZPR-99-358, 1999.
- [BH01] R. Bent and P. Van Hentenryck, *A two-stage hybrid local search for the vehicle routing problem with time windows*, Tech. Report CS-01-06, Brown University, septembre 2001.
- [BHV00] C. Barnhart, C. Hane, and P. Vance, *Using branch-and-price to solve origin-destination integer multicommodity flow problems*, Operations Research **48** (2000), 318–326.
- [BJN<sup>+</sup>98] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance, *Branch-and-price : column generation for solving huge integer programs*, Operations Research **46** (1998), 316–329.
- [BK98] Alexander Bockmayr and Thomas Kasper, *A unifying framework for integer and finite domain constraint programming*, INFORMS Journal on Computing **10** (1998), no. 3, 287–300.
- [CDD<sup>+</sup>99] J.F. Cordeau, G. Desaulniers, J. Desrosiers, M. M. Salomon, and F. Soumis, *The vrp with time windows*, Tech. Report G-99-13, Les Cahiers du GERAD, 1999.

- [CDP02] A. Chabrier, É. Danna, and C. Le Pape, *Coopération entre génération de colonnes et recherche locale appliquée au problème de routage de véhicules*, Journées nationales sur la résolution pratique de problèmes NP-Complets, 2002.
- [CG95] L. Clarke and P. Gong, *Capacited network design with column generation*, 1995.
- [CH01] A. Chabrier and J.-K. Hao, *Prétraitement et réduction de modèles*, RFIA, 2001.
- [Cha99a] A. Chabrier, *A cooperative cp and lp optimizer approach for the pairing generation problem*, CPAIOR, 1999.
- [Cha99b] ———, *Maestro : A column generation modeling and search framework*, Tech. report, ILOG S.A., 1999.
- [Cha99c] ———, *Une approche coopérative ppc - pl pour le problème de génération de pairings*, ROADEF, 1999.
- [Cha00a] ———, *Column generation modeling and search framework*, ISMP, 2000.
- [Cha00b] ———, *Using constraint programming search goals to describe column generation search procedure*, CPAIOR'00, 2000.
- [Cha02a] ———, *Branch-and-price-and-cut to solve a network design problem*, To be published., 2002.
- [Cha02b] ———, *Modèles de génération de colonnes génériques*, ROADEF, 2002.
- [Cha02c] ———, *Vehicule routing problem with elementary shortest path based column generation*, To be published., 2002.
- [Chv] V. Chvatal, *Linear programming*.
- [CLLR01] Y. Caseau, F. Laburthe, C. Le Pape, and B. Rottembourg, *Combining local and global search in a constraint programming environment*, Knowledge Engineering Review **16** (2001), no. 1, 41–68.
- [CLM01] J.-F. Cordeau, G. Laporte, and A. Mercier, *A unified tabu search heuristic for vehicle routing problems with time windows*, Journal of the Operational Research Society **52** (2001), 928–936.
- [CR99] W. Cook and J.L. Rich, *A parallel cutting-plane algorithm for the vehicle routing problem with time windows*, Tech. Report TR99-04, Department of Computational and Applied Mathematics, Rice University, mai 1999.
- [DD86] Y. Dumas and J. Desrosier, *A shortest path problem for vehicle routing with pick-up, delivery, and time windows*, Tech. Report G-86-09, Les Cahiers Du GERAD, 1986.

- [DDE<sup>+</sup>00] G. Desaulniers, J. Desrosiers, A. Erdmann, M. M. Solomon, and F. Soumis, *The vrp with pick-up and delivery*, Tech. Report G-00-25, Les Cahiers Du GERAD, 2000.
- [DDL<sup>M</sup>98] G. Desaulniers, J. Desrosiers, A. Lasry, and M.M.Salomon, *Crew pairing for a regional carrier*, Tech. Report G-98-06, Les Cahiers du GERAD, 1998.
- [DDS01] G. Desaulniers, J. Desrosiers, and M. M. Solomon, *Accelerating strategies in column generation methods for vehicle routing and crew scheduling problems*, vol. Essays and Surveys in Metaheuristics, pp. 309–324, CC. Ribeiro and P. Hansen, editors, Août 2001.
- [Des88] M. Desrochers, *An algorithm for the shortest path problem with resource constraints*, Tech. Report G-88-27, Les cahiers du GERAD, septembre 1988.
- [DFS<sup>+</sup>00] B. De Backer, V. Furnon, P. Shaw, Ph. Kilby, and P. Prosser, *Solving vehicle routing problems using constraint programming and metaheuristics*, Journal of Heuristics **6** (2000), 501–523.
- [dMVDH99] O. du Merle, D. Villeneuve, J. Desrosiers, and P. Hansen, *Stabilized column generation*, Discrete Math. **194** (1999), 229–237.
- [DP98] Z. Degraeve and M. Peeters, *Benchmark results for the cutting stock and bin packing problem*.
- [DS99] Z. Degraeve and L. Schrage, *Optimal integer solutions to industrial cutting stock problems*, INFORMS Journal on Computing **11** (1999), no. 4, 406–419.
- [DV00] S. DeVries and R. Vohra, *Combinatorial auctions : A survey*, 2000.
- [DW60] G.B. Dantzig and P. Wolfe, *Decomposition principle for linear programs*, Operations Research **8** (1960), 101–111.
- [ENNS99] A. Erdmann, A. Nolte, A. Noltemeier, and R. Schrader, *Modeling and solving the airline schedule generation problem*, Tech. Report zpr99-351, University of Cologne, 1999.
- [Erd99] Andreas Erdmann, *Combinatorial optimization problems arising in airline industry*, Ph.D. thesis, Universitat zu Koln, 1999.
- [EY99] P. Eveborn and D. Yuan, *Accelerating column generation through stabilizing in the dual space : A short summary with case*, septembre 1999, Practical Set Partitioning and Column Generation.
- [Far90] A.A. Farley, *A note on rounding a class of linear programming problems, including cutting stock problems*, Operations Research **38** (1990), 922–923.

- [FF58] L.R. Ford and D.R. Fulkerson, *A suggested computation for maximal multi-commodity network flows*, *Management Science* **5** (1958), 97–101.
- [For96] S. Fores, *Column generation approaches to bus driver scheduling*, Tech. report, University of Leeds, School of Computer Studies, 1996.
- [FP98] S. Fores and L. Prool, *Drivers scheduling by integer linear programming - the tracs ii approach*, Tech. Report 98.01, University of Leeds, School of Computer Studies, 1998.
- [FPW96] S. Fores, L. Prool, and A. Wren, *A column generation approach to bus driver scheduling*, Tech. Report 96.22, University of Leeds, School of Computer Studies, 1996.
- [FPW97] ———, *An improved ilp system for driver scheduling*, Tech. Report 97.21, University of Leeds, School of Computer Studies, 1997.
- [GG61] P.C. Gilmore and R.E. Gomory, *A linear programming approach to the cutting-stock problem*, *Operations Research* **9** (1961), 849–859.
- [GH95] M. L. Ginsberg and W. D. Harvey, *Limited discrepancy search*, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*; Vol. 1 (Montréal, Québec, Canada) (Chris S. Mellish, ed.), Morgan Kaufmann, 1995, August 20–25 1995, pp. 607–615.
- [GKM99] V. Gabrel, A. Knippel, and M. Minoux, *Exact solution of multicommodity network optimization problems with general step cost functions*, *Operations Research* **25** (1999).
- [GNS96] Z. Gu, G. Nemhauser, and M. Savelsbergh, *Lifted flow cover inequalities for mixed 0-1 integer programs*, 1996.
- [Gre02] H.J. Greenberg, *Mathematical programming glossary*, World Wide Web, <http://www.cudenver.edu/hgreenbe/glossary/>, 1996–2002.
- [GTA99] L.M. Gambardella, É. Taillard, and G. Agazzi, *MACS-VRPTW : A multiple ant colony system for vehicle routing problems with time windows*, *New Ideas in Optimization* (David Corne, Marco Dorigo, and Fred Glover, eds.), McGraw-Hill, London, 1999, pp. 63–76.
- [Gun98] Oktay Gunluk, *A branch-and-cut algorithm for capacited network design problems*, 1998.
- [HG99] J. Homberger and H. Gehring, *Two evolutionary metaheuristics for the vehicle routing problem with time windows*, *INFOR* **37** (1999), 297–318.
- [HMS01] A. Hadjar, O. Marcotte, and F. Soumis, *A branch-and-cut algorithm for the multiple depot vehicle scheduling problem*, Tech. report, Les Cahiers du GERAD, 2001.

- [ILO02a] ILOG, *Ilog cplex, user's and reference manuals*, Tech. report, ILOG, 2002.
- [ILO02b] ———, *Ilog hybrid optimizers, user's and reference manuals*, Tech. report, ILOG, 2002.
- [ILO02c] ———, *Ilog solver, user's and reference manuals*, Tech. report, ILOG, 2002.
- [JKK<sup>+</sup>99] Ulrich Junker, Stefan E. Karish, Niklas Kohl, Bo Vaaben, Torsten Fahle, and Meinolf Sellmann, *A framework for constraint programming based column generation*, CP99, 1999.
- [JMM98] B. Jaumard, O. Marcotte, and C. Meyer, *Estimation of the quality of cellular networks using column generation techniques*, Tech. Report G-98-02, Les Cahiers du GERAD, Janvier 1998.
- [JMMV99] B. Jaumard, O. Marcotte, C. Meyer, and T. Vovor, *Comparison of column generation models for channel assignment in cellular networks*, 1999.
- [JMV99a] B. Jaumard, C. Meyer, and T. Vovor, *Column/row and elimination methods*, Tech. Report G-99-34, Les Cahiers du GERAD, Juin 1999.
- [JMV99b] ———, *How to combine a column and row generation method with a column or row elimination procedure - application to a channel assignment problem*, Tech. Report G-99-18, Les Cahiers du GERAD, Février 1999.
- [JRT94] M. Junger, G. Reinelt, and S. Thienel, *Practical problem solving with cutting plane algorithms in combinatorial optimization*, Tech. Report ZPR-94-156, Universitat zu Koln, mars 1994.
- [JTa] M. Jünger and S. Thienel, *The abacus system for branch-and-cut-and-price algorithms in integer programming and combinatorial optimization*.
- [JTb] ———, *Basic design ideas for the branch-and-cut-system abacus*.
- [JT97a] ———, *The design of the branch-and-cut system abacus*, Tech. Report ZPR-97-260, Universitat zu Koln, 1997.
- [JT97b] ———, *Introduction to abacus - a branch-and-cut system*, Tech. Report ZPR-97-263, Universitat zu Koln, 1997.
- [KD02] A. Klose and A. Drexl, *A partitioning and column generation approach for the capacited facility location problem*.
- [KLM01] B. Kallehauge, J. Larsen, and O.B.G. Madsen, *Lagrangean duality and non-differentiable optimization applied on routing with time windows - experimental results*, Tech. Report IMM-TR-2001-9, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark, août 2001.

- [Koh95] N. Kohl, *Exact methods for time constraints routing and scheduling problems*, Ph.D. Thesis, Institute of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark (1995).
- [Kor96] R. E. Korf, *Improved limited discrepancy search*, AAAI/IAAI, Vol. 1, 1996, pp. 286–291.
- [KPS00] Ph. Kilby, P. Prosser, and P. Shaw, *A comparison of traditional and constraint-based heuristic methods on vehicle routing problems with side constraints*, Constraints **5** (2000), no. 4, 389–414.
- [Lar99] J. Larsen, *Vehicle routing with time windows - finding optimal solutions efficiently*, <http://citeseer.nj.nec.com/larsen99vehicle.html>, 1999.
- [MG] M. Minoux and M. Gondran, *Graphes et algorithmes*.
- [MM93] T.L. Magnanti and P. Mirchandani, *Shortest paths, single-origin destination network design and associated polyhedra*, Networks **23** (1993), 103–121.
- [MMR93] T.L. Magnanti, P. Mirchandani, and R.Vachani, *The convex hull of two core capacitated network design problems*, Math Programming **60** (1993), 233–250.
- [MMR95] ———, *Modeling and solving the two-facility network loading problem*, Operations Research **43** (1995), 142–157.
- [MMWW99] H. Marchand, A. Martin, R. Weismantel, and L. Wolsey, *Cutting planes in integer and mixed integer programming*, Tech. Report CORE-DP 9953, Universite catholique de Louvain-la-Neuve, 1999.
- [MS98] K. Marriott and P.J. Stuckey, *Programming with constraints : an introduction*, 1998.
- [MT96] A. Mehrotra and M.A. Trick, *A column generation approach for graph coloring*, INFORMS Journal on Computing **8** (1996), no. 4, 344–354.
- [MW98] P. Meseguer and T. Walsh, *Interleaved and discrepancy based search*, European Conference on Artificial Intelligence, 1998, pp. 239–243.
- [OW00] F. Ortega and L. Wolsey, *A branch-and-cut algorithm for the single commodity uncapacitated fixed charge network flow problem*, 2000.
- [PPRS02] C. Le Pape, L. Perron, J.C. Régim, and P. Shaw, *Robust and parallel solving of a network design problem*, CP (2002).
- [PR99] D. Panton and D. Ryan, *Column generation models for optimal workforce allocation with multiple breaks*, 1999.
- [Ree95] C.R. Reeves, *Modern heuristic techniques for combinatorial problems*, 1995.

- [RF81] D.M. Ryan and B.A. Foster, *An integer programming approach to scheduling*, Computer Scheduling of Public Transport Urban Passenger Vehicle and Crew Scheduling, A. WREN (ed.) (1981), 269–280.
- [RGP99] L.-M. Rousseau, M. Gendreau, and G. Pesant, *Using constraint-based operators in a variable neighborhood search framework to solve the vehicle routing problem with time windows*, CP-AI-OR (1999).
- [RGP02] L.M. Rousseau, M. Gendreau, and G. Pesant, *Solving small VRPTWs with constraint programming based column generation*, Proceedings of CP-AI-OR'02, mars 2002, pp. 333–344.
- [RLa] T.K. Ralphs and L. Ladanyi, *Coin/bcp user's manual*.
- [RLb] T.K. Ralphs and L. Ladanyi, *Symphony : A parallel framework for branch, cut, and price*.
- [RLc] T.K. Ralphs and L. Ladányi, *Symphony 2.8 user's manual*.
- [RLTJ] T.K. Ralphs, L. Ladanyi, L.E. Trotter, and Jr., *Branch, cut, and price : Sequential and parallel*.
- [RT95] Y. Rochat and E. Taillard, *Probabilistic diversification and intensification in local search for vehicle routing*, Journal of Heuristics **1** (1995), 147–167.
- [Sav95] M. Savelsbergh, *A branch-and-price algorithm for the generalized assignment problem*, Tech. report, Georgia Institute of Technology, 1995.
- [SD88] M. Solomon and J. Desrosiers, *Time window constrained routing and scheduling problems*, Transportation Science **22** (1988), no. 1.
- [Sha94] D.X. Shaw, *Limited column generation technique for general telecommunication network design problems*, December 1994.
- [Sha98] P. Shaw, *Using constraint programming and local search methods to solve vehicle routing problems*, Principles and Practice of Constraint Programming, 1998, pp. 417–431.
- [SN95] M.W.P. Savelsbergh and G.L. Nemhauser, *A minto short course*, Tech. report, Georgia Institute of Technology, 1995.
- [SN98] ———, *Functional description of minto, a mixed integer optimizer (version 3.0)*, Tech. Report COC-91-03D, Georgia Institute of Technology, 1998.
- [SNS94] M.W.P. Savelsbergh, G.L. Nemhauser, and G.S. Sigismondi, *Minto, a mixed integer optimizer*, Oper. Res. Letters **15** (1994), 47–58.

- [Sol87] M.M. Solomon, *Algorithms for the vehicle routing and scheduling problem with time window constraints*, Operations Research **35** (1987), 254–265.
- [Sou97] F. Soumis, *Decomposition and column generation*, Tech. Report G-97-42, GERAD, June 1997.
- [SS95] M. Sol and M. Savelsbergh, *A branch-and-price algorithm for the pickup and delivery problem with time windows*, Tech. report, Georgia Institute of Technology, 1995.
- [Tai99] E.D. Taillard, *A heuristic column generation method for the heterogeneous vrp*, Operations Research – Recherche opérationnelle **33** (1999), no. 1, 1–14.
- [Tsa93] E. Tsang, *Foundations of constraint satisfaction*, 1993.
- [VAK98] B. Verweij, K. Aardal, and G. Kant, *On an integer multicommodity flow problem from the airline industry*.
- [Van98] P.H. Vance, *Branch-and-price algorithms for the one-dimensional cutting stock problem*, Tech. report, Auburn University, 1998.
- [Van99] F. Vanderbeck, *Computational study of a column generation algorithm for bin packing and cutting stock problems*, Math. Program. **86** (1999), 565–594.
- [Van00] ———, *On dantzig-wolfe decomposition in integer programming and ways to perform branching in a branch-and-price algorithm*, Operations Research **48** (2000), 111–128.
- [VBJ<sup>+</sup>96] P. Vance, C. Barnhart, E. Johnson, G. Nemhauser, A. Krishna, D. Mahidara, and R. Rebello, *A heuristic branch-and-price approach for the airline crew scheduling problem*, 1996.
- [VBJN94] P.H. Vance, C. Barnhart, E.L. Johnson, and G.L. Nemhauser, *Solving binary cutting-stock problems by column generation and branch-and-bound*, Computational Optimization and Applications **3** (1994), 111–130.
- [VBJN95] P. Vance, C. Barnhart, E. Johnson, and G. Nemhauser, *Airline crew scheduling : A new formulation and decomposition algorithm*, Tech. report, School of Industrial and Systems Engineering, Georgia Institute of Technology, 1995.
- [vdAHS] J.M. van den Akker, C.A.J. Hurkens, and M.W.P. Savelsbergh, *Time-indexed formulations for machine scheduling problems : Column generation*.
- [Wal97] T. Walsh, *Depth-bounded discrepancy search*, IJCAI, 1997, pp. 1388–1395.

- [YMdS00] T. H. Yunes, A. V. Moura, and C.C. de Souza, *Hybrid column generation approaches for solving real world crew management problems*, Tech. Report IC-00-18, ???, 2000.
- [Yu98] G. Yu (ed.), *Operations research in the airline industry*, International Series in Operations research and Management Science, vol. 9, Kluwer Academic Publishers, 1998.