N° d'ordre : D.U. 2053
EDSPIC : 489

Université Blaise Pascal - Clermont-Ferrand II

Ecole doctorale
Sciences Pour l'Ingénieur de Clermont-Ferrand

Thèse

présentée par

Hélène TOUSSAINT

pour obtenir le grade de

Docteur d'Université

Spécialité : Informatique

Algorithmique rapide pour les problèmes de tournées et d'ordonnancement

Soutenue publiquement le 23 juillet 2010
devant le jury composé de :

Président :

Dominique FEILLET	Professeur des Universités, Ecole des Mines de Saint-Etienne

Rapporteurs :

Philippe CHRÉTIENNE	Professeur des Universités, Université Pierre et Marie Curie, Paris
Dominique FEILLET	Professeur des Universités, Ecole des Mines de Saint-Etienne
Christian PRINS	Professeur des Universités, Université de Technologie de Troyes

Directeurs de thèse :

Philippe LACOMME	Maître de Conférences, HDR, Université Blaise Pascal, Clermont-Ferrand II
Alain QUILLIOT	Professeur des Universités, Université Blaise Pascal, Clermont-Ferrand II

Time-saving algorithms for vehicle routing and scheduling problems

Thèse préparée au sein du laboratoire LIMOS (Laboratoire d'Informatique de Modélisation et d'Optimisation des Systèmes) Unité Mixte de Recherche 6158 du CNRS .

LIMOS
Complexe scientifique des Cézeaux,
63173 AUBIERE cedex, FRANCE.

Remerciements

Je tiens tout d'abord à remercier mes deux directeurs de thèse, Alain Quilliot et Philippe Lacomme, pour m'avoir encouragée dans cette voie. Ils sont des encadrants exceptionnels de par leurs compétences, leur patience et leur gentillesse.

Je remercie Philippe Chrétienne, Dominique Feillet et Christian Prins pour avoir accepté de participer à mon jury de thèse et pour leurs remarques pertinentes qui m'ont permis d'améliorer la qualité de ce manuscrit.

Je remercie mes collègues et amis (en particulier Jonas, Christine, Andréa, Christophe, Vincent, Ren, Heitor, Raksmey, Jean-Christophe, Patrice, Frédéric, Nathalie, Rose, Martine ... pour ne citer qu'eux) pour leurs conseils et pour les moments de détente qu'on a partagés.

Je tiens également à dire un grand merci à mes parents pour leur soutien permanent et pour l'infinie patience dont ils ont fait preuve à relire ce manuscrit.

Enfin, je tiens à remercier mon conjoint pour m'avoir toujours encouragée et pour le soutien moral qu'il m'a apporté pendant toute la durée de cette thèse.

Résumé

Dans le cadre de cette thèse, nous nous intéressons à la modélisation et à la résolution de différents problèmes de tournées de véhicules et d'ordonnancement. Nous proposons des méthodes approchées qui ont pour but de résoudre les problèmes de manière rapide et efficace. Nous traitons cinq problèmes. Le premier est un problème d'ordonnancement de projet sous contraintes de ressources (RCPSP) que nous résolvons à l'aide d'un multiflot. Nous envisageons également des méthodes de résolution pour des extensions de ce problème (contraintes temporelles ou financières). Le second est un problème de placement en deux dimensions. Nous utilisons une approche originale basée sur sa relaxation en RCPSP. Le troisième est le *Stacker Crane Problem* (SCP). Il fait partie des problèmes de *pickup and delivery*, dans lesquels des marchandises doivent être transportées depuis des origines vers des destinations à l'aide d'une flotte de véhicules. Dans le SCP, un unique véhicule de capacité unitaire est disponible. Nous proposons une résolution originale à base d'arbres pour le cas préemptif. Le quatrième est un problème de transport à la demande avec contraintes financières. Nous résolvons ce problème grâce à une heuristique d'insertion et une technique de propagation de contraintes. Le cinquième mêle problème de tournées et placement en deux dimensions. Il s'agit du 2L-CVRP dans lequel des colis doivent être livrés à des clients. Nous proposons un schéma GRASP×ELS pour ce problème. Des résultats expérimentaux montrent la pertinence des approches proposées.

Mots-clés : complexité, heuristiques, métaheuristiques, ordonnancement, transport, multiflot, placement.

Abstract

In this thesis, we deal with modeling and solving various problems including vehicle routing and scheduling. We propose approximate methods that aim to solve problems quickly and efficiently. Five problems are addressed. The first one is the Resource-Constrained Project Scheduling Problem (RCPSP) for which a multi-commodity flow approach is introduced. We also consider methods dealing with extensions of this problem (financial or time constraints). The second one is a two dimensional Orthogonal Packing Problem (2OPP) with the resolution being based on its RCPSP relaxation. The third one is the Stacker Crane Problem (SCP). It is a pickup and delivery problem : goods must be transported from starting points to various destinations with a fleet of vehicles. In the SCP, a single one-capacity vehicle is available. We propose original tree based models and algorithms for the preemptive case. The fourth one is a Dial-a-Ride Problem (DARP) with financial constraints. We solve this problem through an insertion heuristic using a constraint propagation technique. The fifth one combines routing problem and two dimensional packing. It is the 2L-CVRP in which items have to be delivered to customers. A GRASP×ELS framework is introduced for this problem. Experimental results show the effectiveness of the proposed methods.

Keywords : complexity, heuristics, metaheuristics, scheduling, vehicle routing, multi-commodity flow, packing.

Table des matières

Résumé **7**

Abstract **9**

Introduction **15**

1 Algorithmes et complexité **19**

1.1 Quelques éléments de complexité 19

1.1.1 Algorithmes 19

1.1.2 Complexité et classe d'un problème 21

1.2 Heuristiques et Métaheuristiques 23

1.2.1 Problèmes traités et vocabulaire 23

1.2.2 Les algorithmes gloutons (*greedy algorithms*) 25

1.2.3 Les techniques de base utilisées dans les métaheuristiques 28

1.2.4 Le recuit simulé (*Simulated Annealing* - SA) 31

1.2.5 La recherche tabou (*Tabu Search* - TS) 32

1.2.6 La recherche à voisinage variable (*Variable Neighborhood Search* - VNS) et ses dérivés 32

1.2.7 L'algorithme GRASP (*Greedy Randomized Adaptative Search Procedure*) 34

1.2.8 La recherche locale itérée (*Iterated Local Search* - ILS) 34

1.2.9 La recherche locale évolutionnaire (*Evolutionary Local Search*-ELS) . 35

1.2.10 Hybridation GRASP×ILS et GRASP×ELS 36

1.2.11 Les algorithmes évolutionnaires et mémétiques 37

1.2.12 Scatter Search et Path Relinking 39

1.2.13 L'optimisation par colonies de fourmis (*Ant Colony Optimization - ACO)* 40

1.2.14 Discussion 41

1.2.15 Synthèse sur les métaheuristiques 41

1.3 Un aperçu de deux méthodes exactes 42

1.3.1 L'exploration arborescente 42

1.3.2 La programmation linéaire 48

1.4 Conclusion 50

2 Problèmes d'ordonnancement sous contraintes de ressources 53
2.1 Problèmes d'ordonnancement de projet sous contraintes de ressources 53
2.1.1 Aperçu des problèmes d'ordonnancement 53
2.1.2 Aperçu des études et méthodes existantes pour le RCPSP 56
2.1.3 Définition et modélisation du RCPSP 57
2.1.4 Les techniques classiques de représentation 58
2.1.5 Les techniques classiques de résolution 61
2.2 Proposition de résolution du RCPSP à l'aide des flots 64
2.2.1 Multiflot associé au RCPSP 64
2.2.2 Algorithme d'insertion : génération d'une solution initiale 65
2.2.3 Recherche locale 76
2.2.4 Résultats numériques 78
2.2.5 Conclusion 86
2.3 RCPSP avec *time lags* conditionnels 86
2.3.1 RCPSP avec *time lags* 86
2.3.2 Définition du RCPSP avec *time lags* conditionnels 87
2.3.3 Reformulation du RCPSPTL en un problème de multiflot 88
2.3.4 Algorithme d'insertion : Génération d'une solution initiale 88
2.3.5 Recherche locale 94
2.3.6 Résultats numériques 94
2.3.7 Conclusion 97
2.4 Perspectives : Ajout d'une ressource non renouvelable 97
2.4.1 Notations 98
2.4.2 Adaptation de la technique d'insertion 98
2.4.3 Conclusion 99
2.5 Proposition d'un *double SGS* pour un problème de placement 99
2.5.1 Les problèmes de placement en deux dimensions 99
2.5.2 Le lien entre les problèmes de placement et le RCPSP 101
2.5.3 Résolution du 2OPP 103
2.5.4 Résolution du 2OPP avec rotations 106
2.5.5 Résultats numériques 107
2.5.6 Conclusion 109
2.6 Proposition d'un *double flot* pour un problème de placement 109
2.6.1 Principe général 109
2.6.2 Relation entre double flot et solution du 2OPP 111
2.6.3 Proposition d'un algorithme pour le calcul d'un double flot sans circuit 117
2.6.4 Expérimentations numériques 123
2.6.5 Conclusion sur le double flot 125
2.7 Conclusion 125

3 Deux problèmes de Ramassage et Livraison : *Stacker Crane* et *Dial-a-Ride* 127
3.1 Présentation générale des problèmes de Ramassage et Livraison 127
3.1.1 Problème avec une origine et une destination données pour chaque demande *one-to-one problems*[1-1|-|-] 129

3.1.2 Problèmes avec plusieurs origines et plusieurs destinations (*Many-to-many problems*) [M-M|-|-] . . . 130
3.1.3 Problème du type *one-to-many-to-one pickup and delivery problems* [1-M-1|-|-] . . . 130
3.2 Le *Stacker Crane Problem* Préemptif et Asymétrique . . . 132
3.2.1 Etat de l'art . . . 132
3.2.2 Le *Stacker Crane Problem* Preemptif et Asymétrique (SCPPA) . . . 134
3.2.3 Proposition de modélisation d'une tournée à l'aide d'un arbre . . . 138
3.2.4 Heuristiques à base d'arbres pour le SCPPA . . . 148
3.2.5 Résultats expérimentaux . . . 154
3.2.6 Conclusion . . . 159
3.3 *Dial-a-Ride* avec contraintes financières . . . 159
3.3.1 Le *Dial-a-Ride Problem* - Présentation et état de l'art . . . 159
3.3.2 Définition . . . 161
3.3.3 Vérification des contraintes et évaluation d'une tournée . . . 164
3.3.4 Proposition d'un algorithme d'insertion pour le DARP . . . 167
3.3.5 Ajout des contraintes financières . . . 173
3.3.6 Expérimentations numériques . . . 179
3.3.7 Conclusion . . . 186
3.4 Perspectives : transport à la demande avec correspondances . . . 186
3.4.1 Présentation du problème . . . 186
3.4.2 Modélisation et résolution envisagées . . . 187
3.5 Conclusion . . . 188

4 Problèmes de tournées et d'ordonnancement 189
4.1 Présentation générale des problèmes de VRP . . . 189
4.2 Nouvelle approche pour la résolution du 2L-CVRP . . . 193
4.2.1 Définition du problème et état de l'art . . . 193
4.2.2 Le GRASP×ELS pour le 2|UR|L-CVRP et le 2|UO|L-CVRP . . . 197
4.2.3 Expérimentations numériques . . . 215
4.3 Conclusion . . . 220

Conclusion 225

Notations 229

Index 235

Introduction

Les problèmes d'ordonnancement et de transport jouent un rôle essentiel dans de nombreux secteurs d'activités. D'une manière générale, les problèmes d'ordonnancement consistent à organiser dans le temps ou dans l'espace un ensemble de tâches. On rencontre, par exemple, des problèmes d'ordonnancement dans les systèmes industriels (ateliers, chaînes d'assemblage, gestion de la production ...), dans les systèmes administratifs (élaboration d'emploi du temps, affectation de salles ...) ou encore dans les systèmes informatiques (gestion de processus, grilles de calculs ...). Les problèmes de transport, qui consistent à organiser le transport de biens ou de personnes, ont pris une place importante dans la société. L'organisation du transport se traduit par un problème de tournées de véhicules. Ces problèmes se rencontrent aussi bien dans le cadre de services à la personne (transport de personnes à mobilité réduite, transport scolaire ...) comme dans le cadre de transport de marchandises (livraison de biens, collecte d'ordures ménagères, ...).

Ces deux problèmes comprennent des enjeux financiers : une bonne organisation des tâches ou du transport permet de réduire les coûts et d'augmenter les profits. C'est pourquoi, ces problèmes intéressent aussi bien les chercheurs que les chefs d'entreprises.

Les problèmes d'ordonnancement et de tournées sont des problèmes combinatoires, c'est-à-dire des problèmes pour lesquels il existe un grand nombre de solutions. Analyser chacune des solutions pour trouver la meilleure n'est donc, en général, pas possible. Dans cette thèse nous proposons d'étudier plusieurs problèmes d'ordonnancement et de tournées, et de les résoudre grâce à des méthodes approchées (heuristiques). Ces méthodes ne garantissent pas l'optimalité de la solution mais permettent d'approcher une solution de bonne qualité, pour des problèmes de grande taille, dans des temps de calcul raisonnables.

Cette thèse se compose de quatre chapitres.

Le premier chapitre introduit les notions d'algorithmes et de complexité et présente quelques méthodes générales de résolution pour les problèmes difficiles. Ce chapitre a pour objectif de donner les bases nécessaires à la compréhension des problématiques rencontrées lors de la résolution de problèmes combinatoires. Il se veut accessible et généraliste.

Nous définissons, tout d'abord, les notions fondamentales liées à la complexité d'un problème comme la décidabilité ou la NP-complétude. Nous nous intéressons, ensuite, aux méthodes de résolution pour les problèmes difficiles. La présentation des principales heuristiques et métaheuristiques constitue la majeure partie de ce chapitre. Il est rappelé le vocabulaire et les procédés indispensables à ces méthodes (optimum, optima locaux, transformation locale, voisinage, recherche locale ...). Enfin, deux méthodes exactes sont pré-

sentées : l'exploration arborescente et la programmation linéaire. De nombreux exemples, appliqués au problème NP-complet du voyageur de commerce, illustrent les méthodes présentées. Cela permet de mettre en évidence la complexité du problème et des méthodes ainsi que les problématiques associées.

Le second chapitre s'intéresse à des problèmes de type ordonnancement : plus précisément il s'agit de problème d'ordonnancement de projet et de placement en deux dimensions. Nous commençons par donner un aperçu des problèmes classiques en ordonnancement de projet et en ordonnancement d'atelier afin de situer parmi ceux-ci le principal problème de ce chapitre : le RCPSP (*Resource-Constrained Project Scheduling Problem*). Le RCPSP est un problème d'ordonnancement de projet sous contraintes de ressources. Nous proposons, pour ce problème, un modèle et des algorithmes de résolution basés sur un multiflot. Ce modèle est ensuite étendu à une extension du RCPSP qui inclut des contraintes temporelles particulières nommées « time lags conditionnels ». Nous montrons ainsi que le modèle que nous proposons est facilement adaptable. Nous évoquons la possibilité de traiter d'autres extensions comme la prise en compte de contraintes financières. Nous montrons ensuite comment les méthodes de résolution pour le RCPSP peuvent être avantageusement utilisées dans le cadre d'un problème de placement : le 2OPP (*Two Orthogonal Packing Problem*). Le 2OPP est un problème de placement orthogonal en deux dimensions. Deux méthodes sont proposées pour résoudre ce problème : la première s'appuie sur un schéma de génération d'ordonnancements, la seconde tire profit du formalisme des flots.

Le troisième chapitre aborde deux problèmes de ramassage et livraison : le *Stacker Crane Problem* et le *Dial-a-Ride Problem*. Les problèmes de ramassage et livraison sont mieux connus sous le nom de problèmes de *Pickup and Delivery* (PDP). Ce chapitre commence par un aperçu général des différents problèmes de *Pickup and Delivery*. Cela permet de situer les problèmes auxquels nous nous intéressons parmi ceux-ci. Le *Stacker Crane Problem* (SCP) est inspiré du fonctionnement des grues portiques. Nous l'étudions sous sa forme préemptive et asymétrique. Le *Dial-a-Ride Problem* (DARP) est un problème de transport à la demande. Il met l'accent sur la qualité de service. Dans ce chapitre, ce problème est étudié avec une contrainte financière supplémentaire.

Le quatrième et dernier chapitre présente un problème récent de tournées de véhicules : le 2L-CVRP (*Two dimensional Loading Capacitated Vehicle Routing Problem*). Une synthèse des différents problèmes classiques de tournées de véhicules est proposée au début du chapitre. Le 2L-CVRP est particulier car, contrairement aux problèmes classiques de tournées de véhicules, il inclut une contrainte de placement. Le 2L-CVRP consiste à livrer des colis à des clients à l'aide d'une flotte de véhicules de capacité limitée. Ainsi, il inclut un problème de chargement en deux dimensions de type 2OPP. Les résultats du chapitre 2 sont donc utilisés.

Nous terminons par une conclusion qui reprend les travaux réalisés et qui dégage les perspectives de recherche.

Dans cette thèse, les problèmes sont identifiés par leur nom anglais. Lorsqu'il existe,

l'équivalent français est également donné. Cependant, les abréviations correspondent aux termes anglais, car il s'agit des acronymes les plus utilisés et les plus connus.

Les travaux présentés dans cette thèse ont fait l'objet des publications suivantes :

- deux articles en révision pour des revues : [KLQT10c] (pour RAIRO-Operations Research) et [DLQT10b] (pour Computer & Operations Research) ;
- cinq publications dans des conférences : [DLQT10a], [KLQT10a], [QT10b], [KLQT10b] et [DLQT09] ;
- une présentation dans le Groupe de Travail Transport-Logistique (GT2L) le 18 juin 2009 à L'Ecole des Mines de Nantes (`http://losi.utt.fr/fr/gt2l.html`).

Chapitre 1

Algorithmes et complexité

Ce chapitre aborde dans un premier temps des notions générales sur les algorithmes et la complexité, puis introduit quelques grands schémas algorithmiques que nous appliquons à la résolution de problèmes d'ordonnancement et de tournées dans les chapitres suivants. Il a pour but de rappeler le vocabulaire et les principes généraux, et non de reformuler la théorie de la complexité d'un point de vue formel ou de donner une liste exhaustive de tous les schémas algorithmiques existants.

1.1 Quelques éléments de complexité

Le but de la théorie de la complexité est de fournir des outils pour l'évaluation, a priori et a posteriori des performances des algorithmes et de la difficulté des problèmes. Elle permet d'estimer, de manière théorique, le temps et les besoins en mémoire pour résoudre un problème. Afin de présenter plus précisément la notion de complexité, il nous faut introduire les notions d'algorithme et de problème.

1.1.1 Algorithmes

Une définition formelle de la notion d'algorithme a été introduite par Alan Turing en 1936 (voir [Tur36]), cette définition s'appuie sur ce qui s'appelle aujourd'hui les *machines de Turing*. De nombreux ouvrages ([CGH96] par exemple) utilisent les machines de Turing déterministes pour introduire la notion de complexité d'un algorithme. Nous nous contentons ici de donner une idée générale sur les notions d'algorithme et de complexité et n'utilisons donc pas le formalisme des machines de Turing. Nous définissons un algorithme de manière informelle comme *une suite d'instructions élémentaires destinée à résoudre un problème.*

Un tel problème $\mathcal{P}$ est défini par :

- Un objet e en entrée de type E (où E est l'ensemble des entrées possibles pour le problème) ;
- Un objet s en sortie de type S (où S est l'ensemble des sorties possibles pour le problème) ;
- Un ensemble de relations $\mathcal{R}$ décrivant le lien qui doit exister entre e et s.

On appelle instance et on note $\mathcal{I}$ un jeu de données (c'est-à-dire des valeurs) pour e. On notera $\mathcal{R}(\mathcal{I}, s) = 1$ si s et $\mathcal{I}$ sont effectivement liés par l'ensemble de relations $\mathcal{R}$ et

$\mathcal{R}(\mathcal{I}, s) = 0$ sinon. Résoudre $\mathcal{P}$ revient alors, étant donné un objet $e \in E$, une instance $\mathcal{I}$ de e, à trouver un objet $s \in S$ tel que $\mathcal{R}(\mathcal{I}, s) = 1$.

Exemple 1. *Eléments définissant le problème de tri d'un tableau.*

- $E =$ *l'ensemble des tableaux ;*
- $e =$ *un tableau (par exemple* $\begin{array}{|c|c|c|c|c|c|}\hline \ & \ & \ & \ & \ & \ \\ \hline\end{array}$*) ;*
- $\mathcal{I} =$ *un jeu de données pour le tableau e (par exemple* $\begin{array}{|c|c|c|c|c|c|}\hline 12 & 6 & 8 & 2 & 3 & 9 \\ \hline\end{array}$*);*
- $S =$ *l'ensemble des tableaux ;*
- $s =$ *un tableau ;*
- $\mathcal{R}(e, s) = 1$ *si et seulement si e et s contiennent exactement les mêmes valeurs et ces valeurs sont triées dans le tableau s (ici* $\mathcal{R}(e, s) = 1$ *ssi* $s = \begin{array}{|c|c|c|c|c|c|}\hline 2 & 3 & 6 & 8 & 9 & 12 \\ \hline\end{array}$*).*

Un algorithme $\mathcal{A}$ associé à un problème $\mathcal{P}$ calcule, à partir de e, un objet s de telle sorte que l'ensemble des relations liant e et s soient respectées. Notons que dans le cas où l'ensemble des relations liant e et s est complexe, il est possible que l'algorithme $\mathcal{A}$ ne trouve pas de solution s telle $\mathcal{R}(\mathcal{I}, s) = 1$ et génère alors une solution avec un certain niveau d'erreur.

On note $s_{\mathcal{A}}$ la solution calculée par l'algorithme $\mathcal{A}$ à partir d'une instance $\mathcal{I}$ et on note s^* un élément de S tel que $\mathcal{R}(\mathcal{I}, s^*) = 1$.

On peut alors mesurer la performance de $\mathcal{A}$ pour l'instance $\mathcal{I}$ selon différents critères :

le temps d'exécution : il représente le temps nécessaire à $\mathcal{A}$ pour calculer $s_{\mathcal{A}}$ à partir de $\mathcal{I}$;

l'espace mémoire : il représente la quantité de mémoire nécessaire à $\mathcal{A}$ pour calculer $s_{\mathcal{A}}$ à partir de $\mathcal{I}$;

l'erreur : elle représente l'erreur éventuelle qui peut exister entre $s_{\mathcal{A}}$ et s^*. Notons qu'une telle erreur ne peut être mesurée que si l'on possède une métrique sur l'espace des objets de S et que l'on est capable de mesurer la distance entre $s_{\mathcal{A}}$ et s^* ;

la robustesse : elle exprime la capacité de $\mathcal{A}$ à s'adapter à des variations sur $\mathcal{I}$ (de manière intuitive, un algorithme est robuste si un faible changement dans les données n'induit pas ou peu de changements dans la solution, au contraire si la solution doit être entièrement recalculée alors l'algorithme n'est pas robuste).

Pour les études de complexité, il est nécessaire de définir la taille d'une instance. De manière gérérale, on dira qu'une instance est de taille n si l'instance occupe n cases mémoire. La complexité d'un algorithme est alors une fonction de n.

On définit la complexité $\mathcal{C}_{\mathcal{A}}$ d'un algorithme $\mathcal{A}$ comme le nombre maximal d'instructions élémentaires effectuées par $\mathcal{A}$ pour traiter n'importe quelle instance, de taille n fixée, d'un problème. Il en découle les notions de polynomialité et d'exponentialité : $\mathcal{A}$ est dit polynomial, s'il existe un polynôme P tel que : $\forall n \in \mathbb{N}$, $\mathcal{C}_{\mathcal{A}}(n) \leq P(n)$. Dans le cas contraire $\mathcal{A}$ est dit exponentiel. Remarquons qu'il n'est nullement besoin que $\mathcal{C}_{\mathcal{A}}$ s'exprime par un polynôme pour que l'algorithme soit polynomial, il suffit que $\mathcal{C}_{\mathcal{A}}$ soit majorée par un polynôme. De même un algorithme peut être exponentiel sans que sa complexité s'exprime par une exponentielle.

Cette mesure de complexité est dite *complexité au plus mauvais cas* puisque parmi toutes les instances de taille n c'est l'instance qui engendre le plus d'instructions qui détermine la complexité de l'algorithme. Cette mesure est pratique au niveau des calculs théoriques. Cependant, il arrive qu'en pratique la complexité mesurée d'un algorithme soit totalement différente de sa complexité théorique au plus mauvais cas. Un des exemples les plus connus et les plus révélateurs est celui de l'algorithme du simplexe pour la programmation linéaire. Cet algorithme prend en entrée une matrice de contraintes de taille $n_1 \times n_2$ et il est possible de construire des instances qui vont requérir une exécution de l'ordre de 2^{n_1}. Cependant, on s'aperçoit que la complexité de cet algorithme est en moyenne de l'ordre de $n_1 + n_2$. On peut alors introduire une autre mesure de la complexité comme étant le nombre moyen d'instructions élémentaires effectuées par $\mathcal{A}$ pour traiter toutes les instances de taille n d'un problème. Cependant, une telle définition est difficile à manipuler, entre autre à cause des calculs de probabilité qu'elle induit.

Ces difficultés liées à la gestion d'outils de mesure de performance a priori a pour conséquence que l'essentiel du processus d'évaluation doit avoir lieu a posteriori, par le biais de tests. Cependant construire des procédures de tests significatifs et rigoureux constitue souvent un problème délicat. Le théorème de Fagin [Fag74] fait apparaître que, pour la plupart des problèmes d'optimisation combinatoire à réponse binaire 0/1 (les problèmes d'existence par exemple), une génération d'instances à l'aide d'un processus aléatoire apparemment neutre va induire une convergence presque sûrement vers 1 ou vers 0.

La complexité d'un algorithme s'exprime souvent par un ordre de grandeur dépendant de la taille n des données :

$O(1)$: complexité constante (indépendante de la taille de la donnée)
$O(\log(n))$: complexité logarithmique
$O(n)$: complexité linéaire
$O(n^p)$: complexité polynomiale ($p \in \mathbb{N}$, $p \geq 2$)
$O(a^n)$: complexité exponentielle ($a \in \mathbb{R}$, $a > 1$)

1.1.2 Complexité et classe d'un problème

La théorie de la complexité permet de classer les problèmes et d'établir des relations d'inclusion parmi ces classes, que l'on appelle *classes de complexité*. La complexité d'un problème est donnée par rapport à la plus faible complexité de l'algorithme qui résout ce problème. Notons que cette approche est délicate puisqu'il peut exister plusieurs algorithmes, éventuellement de complexité différente, pour résoudre un même problème, c'est pourquoi on choisit l'algorithme de plus faible complexité. Nous donnons ici un aperçu général des grandes classes de complexité qui concernent les problèmes de décision (pour une approche plus formelle voir [CGH96]).

1.1.2.1 Problèmes décidables

Un problème $\mathcal{P}$ est décidable, s'il existe un algorithme qui résout $\mathcal{P}$ en un temps fini. Dans le cas inverse $\mathcal{P}$ est dit indécidable. Parmi les problèmes décidables, on peut faire apparaître trois classes bien connues : la classe P, la classe *P-Espace* et la classe NP.

1.1.2.2 La classe P

Un problème $\mathcal{P}$ est dit polynomial (ou polynomial-temps), s'il existe un algorithme de complexité polynomiale qui résout $\mathcal{P}$. L'ensemble des problèmes polynomiaux forme la classe P.

1.1.2.3 La classe P-*Espace*

Un problème $\mathcal{P}$ est dit polynomial-espace, s'il existe un algorithme qui résout $\mathcal{P}$ en espace polynomial par rapport à la taille de ses données.

1.1.2.4 La classe NP

Un problème $\mathcal{P}$ est dans NP si pour toute instance de ce problème on peut vérifier que l'instance est solution du problème en temps polynomial.

Cette catégorie est fondamentale car elle contient la plupart des problèmes d'optimisation combinatoire. Remarquons que, pour n'importe quelle instance d'un problème polynomial, il est toujours possible de vérifier en temps polynomial si elle est solution de ce problème, donc $P \subseteq NP$. La question de savoir si $P = NP$ est, à ce jour, une question ouverte. C'est pourquoi, on s'intéresse particulièrement aux problèmes que l'on peut considérer comme « les plus difficiles » de la classe NP au sens où, la découverte d'un algorithme polynomial pour résoudre un tel problème entraînerait l'existence d'algorithmes polynomiaux pour résoudre n'importe quel problème de NP. Un tel problème est qualifié de NP-*complet*.

1.1.2.5 Les problèmes NP-*complet*

Une notion essentielle pour définir ce qu'est un problème NP-*complet* est celui de la dominance : Un problème $\mathcal{P}$ domine un problème $\mathcal{P}'$ (ou encore $\mathcal{P}'$ est réductible polynomialement en $\mathcal{P}$) si :

1. On peut transformer toute instance $\mathcal{I}$ de $\mathcal{P}$ en une instance $\mathcal{I}'$ de $\mathcal{P}'$ grâce à un algorithme polynomial ;
2. $\mathcal{I}$ est solution de $\mathcal{P}$ si et seulement si $\mathcal{I}'$ est solution de $\mathcal{P}'$.

Il découle de cette définition qu'un problème NP-*complet* est un problème de la classe NP qui domine tous les autres. La notion de NP-*complétude* est donc associée à une notion de dominance et non, comme il est parfois dit abusivement, à une notion d'exponentialité. Le schéma 1.1 synthétise les relations d'inclusion entre les classes P, NP et P-*Espace*.

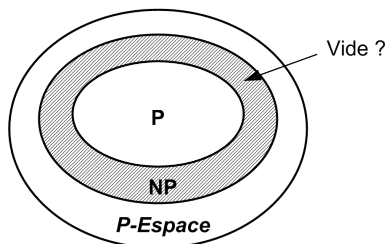


FIG. 1.1 – Sens des inclusions entre les classes P, NP et P-*Espace*

1.1.2.6 Un exemple de problème *NP-complet*

Nous introduisons ici un exemple classique en optimisation combinatoire, celui du voyageur de commerce (*Traveling Salesman Problem* - TSP). Le problème du voyageur de commerce consiste, étant donné un ensemble de villes séparées par des distances, à trouver le plus court chemin qui relie toutes ces villes. Ce chemin doit contenir une et une seule fois chaque ville. La ville de départ n'a pas d'importance puisque la solution définit un cycle : une permutation circulaire d'une solution donne la même solution ($ABCDEF$ est identique à $DEFABC$). Ce problème est *NP-complet*.

Comme on l'a dit précédemment, la question de savoir si $P = NP$ est une question ouverte, pour le moment on ne sait pas résoudre les problèmes *NP-complet* en temps polynomial. Afin de résoudre ces problèmes, on utilise différents types d'algorithmes. On peut en citer par exemple trois :
- les algorithmes approchés (appelés heuristiques), qui permettent de trouver des solutions approchées dans des temps de calcul raisonnables, mais sans avoir d'indication sur la qualité de la solution trouvée ;
- les algorithmes d'approximation avec garantie, qui permettent de trouver des solutions approchées et garantissent une qualité de la solution fournie (sous forme d'écart maximal à la solution optimale) pour toutes les instances du problème ;
- les algorithmes exacts, comme l'exploration arborescente ou la programmation linéaire qui, couplées avec des mécanismes de filtrage, peuvent s'avérer très performantes. Ces méthodes fournissent des solutions exactes mais le temps de calcul n'est pas borné polynomialement. C'est pourquoi, elles sont souvent réservées à des instances de taille modérée.

Dans la seconde section de ce chapitre, nous donnons un aperçu des heuristiques et métaheuristiques classiques, puis dans la troisième section nous nous intéressons aux algorithmes exacts. Nous abordons, en particulier, les techniques d'exploration arborescente et de programmation linéaire.

1.2 Heuristiques et Métaheuristiques

Dans cette section, nous nous intéressons aux méthodes de résolution approchées. Nous commençons par décrire les problèmes auxquels elles s'adressent ainsi que leur points clés. Nous présentons ensuite, les principales méthodes existantes.

1.2.1 Problèmes traités et vocabulaire

On s'intéresse dans cette partie à des problèmes d'optimisation. Un tel problème est défini par un ensemble de contraintes et une fonction coût f à minimiser. Notons Ω l'ensemble des objets respectant les contraintes du problème (Ω est un ensemble fini mais de cardinal souvent très grand). On peut alors formuler le problème de la manière suivante :
« Trouver $\mathcal{S}$ dans Ω tel que $\mathcal{S}$ minimise f », ou encore sous sa forme mathématique :

$$\underset{\mathcal{S}\in\Omega}{\text{Minimiser}}\ f(\mathcal{S})$$

L'espace Ω est appelé « espace des solutions » ou « espace de recherche ». Un élément de Ω s'appelle solution réalisable (ou plus simplement solution) et l'élément de Ω qui minimise f s'appelle solution optimale.

Ce type de problème peut être résolu grâce à des méthodes exactes (programmation linéaire, exploration arborescente ...) mais seulement pour des instances de taille modérée, les temps de calcul devenant vite démesurés pour des instances de plus grande taille. On est donc souvent amené à les résoudre de manière approchée et on recherche alors non pas la solution optimale mais une solution de bonne qualité dans des temps de calcul raisonnables. Les notions de « bonne qualité » et de « temps de calcul raisonnable » étant assez subjectives, le compromis temps de calcul / qualité de la solution est souvent difficile à déterminer. Nous présentons ici des algorithmes de résolution approchée que l'on appelle heuristiques et métaheuristiques. Il n'existe pas, à notre connaissance, de définition officielle, cependant il est communément admis que :

- une heuristique désigne un algorithme qui résout un problème d'optimisation donné, sans garantie d'optimalité mais dans des temps de calcul raisonnables (un exemple connu est l'algorithme glouton, voir section 1.2.2) ;
- une métaheuristique désigne un schéma algorithmique général qui peut s'appliquer à différents problèmes d'optimisation combinatoire. Plus précisément, elle utilise des stratégies qui guident la recherche dans l'espace des solutions, ces stratégies étant indépendantes du problème auquel on les applique. Le but est d'explorer le plus efficacement possible l'espace des solutions afin de ne pas rester bloqué dans les minima locaux et de se diriger rapidement vers les régions les plus prometteuses. Il existe un grand nombre de métaheuristiques allant de schémas très simples (qui mettent en œuvre des processus de recherche basiques, comme la descente, voir section 1.2.3.3), à des schémas beaucoup plus complexes (avec des processus de recherche élaborés comme les colonies de fourmis, voir section 1.2.13).

Les métaheuristiques sont devenues des méthodes très populaires pour la résolution de problèmes d'optimisation combinatoire, c'est pourquoi il existe de nombreux ouvrages qui leur sont consacrés. On peut citer, entre autre, [BR03] qui donne une vue d'ensemble des métaheuristiques et propose une classification, ou encore [DPST03] qui détaille les principales métaheuristiques et présente des études de cas. La majorité des ouvrages consacrés aux métaheuristiques s'adresse à un public déjà familier avec les problématiques liées à la résolution de problèmes d'optimisation combinatoire. Par conséquent, ils abordent rapidement les points techniques des métaheuristiques sans rappeler, ou de manière très brève, les notions et le vocabulaire élémentaires (voisinage, recherche locale, optimum local, optima locaux ...) qui sont supposés acquis par le lecteur. Contrairement à ceux-ci, le but de cette section n'est pas de décrire de manière très technique les différentes métaheuristiques mais de sensibiliser le lecteur à ces méthodes, au vocabulaire et aux problématiques de base dans la résolution heuristique des problèmes d'optimisation combinatoire. Nous donnons un aperçu des métaheuristiques les plus connues en insistant plus particulièrement sur la stratégie de recherche qu'elles utilisent.

Le vocabulaire de base, ci-dessous, s'appuie sur la définition du problème de minimi-

sation donnée au début de cette section et il est détaillé à travers des exemples dans les sections 1.2.2 et 1.2.3 :

Transformation locale Une transformation locale $\mathcal{T}$ est une opération qui modifie légèrement une solution $\mathcal{S}$ de Ω en une solution $\mathcal{S}'$.

Voisinage Etant donnée une transformation locale $\mathcal{T}$, on appelle voisinage d'une solution $\mathcal{S}$, et on note $\mathcal{V}(\mathcal{S})$, l'ensemble des solutions de Ω qu'il est possible d'obtenir en appliquant $\mathcal{T}$ à $\mathcal{S}$. Notons que le voisinage d'une solution dépend complètement de la transformation locale utilisée.

Optimum global L'optimum global d'un problème désigne sa solution optimale : c'est la solution de plus faible coût dans tout Ω.

Optima locaux On appelle optimum local une solution localement optimale : c'est la solution de plus faible coût dans une région donnée de Ω : $\mathcal{S}$ est un optimum local (pour une topologie de voisinage donnée) si pour tout voisin $\mathcal{S}'$ de $\mathcal{S}$ on a $f(\mathcal{S}) \leq f(\mathcal{S}')$. La figure 1.2 schématise l'allure de la fonction objectif f dans l'espace Ω, elle situe l'optimum global et les optima locaux.

Recherche locale On appelle recherche locale le processus qui consiste, à partir d'une solution courante $\mathcal{S}$, à explorer partiellement l'espace Ω grâce à des transformations locales de manière à améliorer $\mathcal{S}$. Ce processus mène généralement à un optimum locale.

Bassin d'attraction On appelle bassin d'attraction d'un minimum local $\mathcal{S}^*_{loc}$ l'ensemble des solutions $\mathcal{S}$ telles que une recherche locale à partir de $\mathcal{S}$ mène à $\mathcal{S}^*_{loc}$.

Intensification (ou exploitation) et diversification (ou exploration) Il existe plusieurs stratégies de recherche de solutions dans Ω. On utilise le terme intensification lorsqu'on se sert de l'expérience et de la connaissance déjà acquises pour guider la recherche. On utilise le terme diversification lorsqu'on explore l'espace de recherche pour acquérir de nouvelles informations.

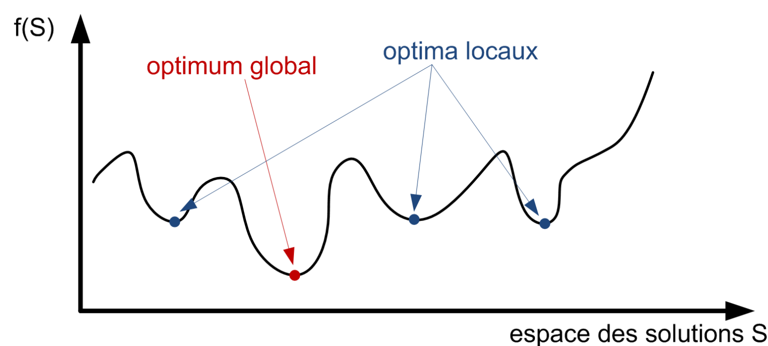


Fig. 1.2 – Allure de la fonction objectif dans l'espace des solutions

1.2.2 Les algorithmes gloutons (*greedy algorithms*)

Les algorithmes gloutons (greedy algorithms) sont parmi les schémas heuristiques les plus simples et les plus rapides. Ils construisent une solution de manière itérative sans

jamais remettre en cause les décisions prises à l'itération antérieure. Ces algorithmes construisent une solution élément par élément. A une itération donnée, on détermine l'élément à inclure dans la solution partielle en évaluant le coût de la nouvelle solution partielle qui inclut cet élément. L'élément engendrant la plus petite augmentation du coût est choisi (voir algorithme 1). Notons que, le cas (rare) où l'ensemble des solutions du problème traité a une structure de matroïde conduit à des algorithmes gloutons exacts. Cependant, dans le cas général, ces algorithmes sont approchés et les insertions qui semblent les meilleures à une itération donnée peuvent s'avérer, en fait, inappropriées pour la suite du processus. Malheureusement, il n'est pas possible de connaître, a priori, l'impact des décisions prises à une itération donnée sur le long terme. En outre, il est possible qu'à une itération donnée aucun élément ne soit insérable (par exemple à cause de contraintes qui se retrouvent violées), dans ce cas, l'algorithme échoue.

Algorithme 1 : L'algorithme glouton déterministe

Sorties : $\mathcal{S}$ ou echec

```
1 echec ← faux;
2 S ← ∅ (objet vide);
3 Tant que (S incomplète et echec = faux) faire
4 |   Construire la liste L des éléments insérables dans S;
5 |   Si L ≠ ∅ alors
6 |   |   Evaluer le coût incrémental des éléments de L;
7 |   |   Insérer dans S l'élément ayant le coût incrémental le plus faible
8 |   sinon
9 |   |   echec ← vrai;
```

Un algorithme glouton peut facilement être randomisé en choisissant à chaque itération, non pas le meilleur élément, mais un élément au hasard parmi les k meilleurs (k étant un paramètre à fixer). En pratique on remarque que cette approche peut générer des solutions de coûts très différents. L'idée est alors d'exécuter l'algorithme randomisé un grand nombre de fois et de conserver la meilleure solution obtenue.

Exemple 2. *Illustration de l'algorithme glouton sur le problème du voyageur de commerce.*

Nous utilisons pour cet exemple une instance à 5 villes, la distance entre chaque ville est donnée dans le tableau 1.1. Rappelons que l'objectif de ce problème est de trouver dans quel ordre il faut parcourir les villes pour minimiser la distance totale parcourue par le voyageur, sachant qu'il doit revenir à sa ville de départ. Une solution est appelée tournée, elle est notée sous forme de liste $(v_0, \ldots, v_n)$, où v_i désigne une ville, v_{i+1} désigne la ville visitée directement après v_i et $v_0 = v_n$. La distance à parcourir pour effectuer cette tournée est appelée coût de la tournée.

Supposons, pour notre exemple, que la ville de départ soit la ville 0. On considère alors que la tournée vide (ou tournée triviale) est (0,0), son coût est 0. Il faut ajouter itérativement les villes 1, 2, 3 et 4 dans cette tournée. On évalue à chaque itération, pour chacune des villes non encore insérées et pour chaque emplacement possible dans la tournée partielle, le coût engendré par l'insertion envisagée :

	0	1	2	3	4
0	0	25	79	20	75
1	25	0	75	45	96
2	79	75	0	80	125
3	20	45	80	0	55
4	75	96	125	55	0

TAB. 1.1 – Exemple d'instance du voyageur de commerce (distances entre chaque ville)

itération 1 On évalue le coût des tournées (0,1,0), (0,2,0), (0,3,0), (0,4,0). On trouve respectivement 50, 158, 40, 150. La ville 3 est donc choisie, c'est elle qui engendre le plus petit coût, on a donc au terme de l'itération 1 la tournée partielle $\mathcal{S} = (0, 3, 0)$.

itération 2 On envisage les tournées (0,1,3,0), (0,3,1,0), (0,2,3,0), (0,3,2,0), (0,4,3,0), (0,3,4,0). Par symétrie, le coût de (0,1,3,0) est le même que celui de (0,3,1,0) et il en va de même pour les tournées suivantes deux à deux. Il suffit donc d'évaluer (0,1,3,0), (0,2,3,0), (0,4,3,0). On trouve respectivement 90, 179 et 150. On choisit d'insérer la ville 1 entre 0 et 3, on obtient (0,1,3,0).

itération 3 On envisage les tournées (0,2,1,3,0), (0,1,2,3,0), (0,1,3,2,0), (0,4,1,3,0), (0,1,4,3,0), (0,1,3,4,0). Les coûts respectifs sont 219, 200, 229, 236, 190, 200. On choisit d'insérer 4 entre 1 et 3, on obtient (0,1,4,3,0).

itération 4 On envisage les tournées (0,2,1,4,3,0), (0,1,2,4,3,0), (0,1,4,2,3,0) et (0,1,4,3,2,0). Les coûts respectifs sont 325, 300, 346 et 335. On conserve (0,1,2,4,3,0) qui a le coût le plus faible (300). Toutes les villes ont été insérées, l'algorithme s'arrête et retourne la solution $\mathcal{S}$ = (0,1,2,4,3,0) qui s'avère être la solution optimale (sur des instances un peu plus complexes les algorithmes gloutons trouvent rarement la solution optimale). Une illustration pour cet exemple des différentes étapes de l'algorithme est donnée figure 1.3.

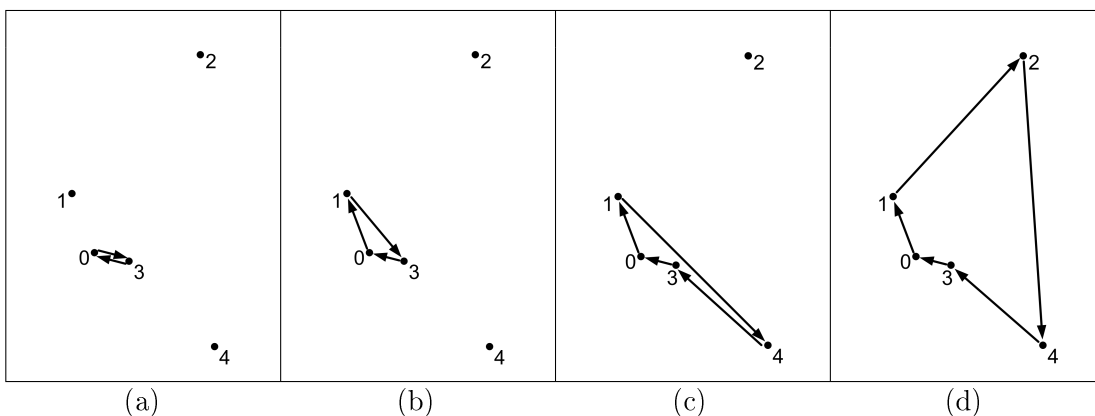


FIG. 1.3 – Illustration de l'exemple 2 :(a) tournée partielle après l'itération 1 ; (b) tournée partielle après l'itération 2 ; (c) tournée partielle après l'itération 3 ; (d) tournée finale

Ces algorithmes de construction peuvent donner des solutions de qualité peu satisfaisante, mais ils sont très utiles pour initialiser des algorithmes basés sur les stratégies de recherche dans l'espace des solutions comme les métaheuristiques. Pour bien comprendre

le fonctionnement des métaheuristiques, il est nécessaire que les notions d'opérateur de transformation locale, de voisinage et de recherche locale soient suffisamment claires. Nous allons donc les illustrer sur un exemple dans la section suivante.

1.2.3 Les techniques de base utilisées dans les métaheuristiques

1.2.3.1 Transformation locale appliquée à une solution du voyageur de commerce

Une transformation locale est une opération qui modifie légèrement une solution. Il existe diverses transformations locales applicables à une solution du voyageur de commerce. Nous allons en présenter deux : l'échange et le 2-opt. Considérons une instance à 6 villes et la solution (0,5,3,1,4,2,0).

- Echange : on échange deux villes aléatoirement, par exemple 5 et 4, on obtient (0,4,3,1,5,2,0) ;
- 2-opt : on échange deux arcs (voir figure 1.4), par exemple 5-3 et 4-2, on obtient (0,5,4,1,3,2,0).

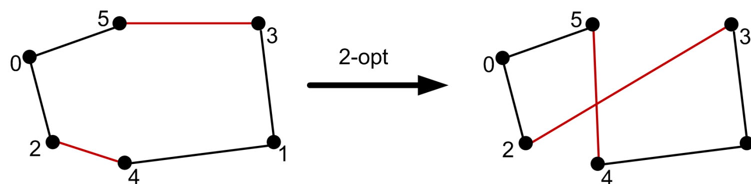


Fig. 1.4 – Utilisation de la transformation locale 2-opt : on échange l'arc 5-3 et l'arc 4-2

1.2.3.2 Voisins et voisinage d'une solution

Le voisinage d'une solution $\mathcal{S}$ est l'ensemble des voisins de $\mathcal{S}$, c'est-à-dire l'ensemble des solutions que l'on peut obtenir à partir de $\mathcal{S}$ en utilisant une transformation locale donnée. Considérons la solution précédente $\mathcal{S} = (0,5,3,1,4,2,0)$. Le voisinage de cette solution par rapport à la transformation locale « Echange » est $\mathcal{V}(\mathcal{S}) = \{$ (0,3,5,1,4,2,0) ; (0,1,3,5,4,2,0) ; (0,4,3,1,5,2,0) ; (0,2,3,1,4,5,0) ; (0,5,1,3,4,2,0) ; (0,5,4,1,3,2,0) ; (0,5,2,1,4,3,0) ; (0,5,3,4,1,2,0) ; (0,5,3,2,4,1,0) ; (0,5,3,1,2,4,0) $\}$ dans lesquels ont été échangés respectivement 5 et 3 ; 5 et 1 ; 5 et 4 ; 5 et 2 ; 3 et 1 ; 3 et 4 ; 3 et 2 ; 1 et 4 ; 1 et 2 ; 4 et 2.

1.2.3.3 Recherche locale

Une recherche locale consiste, à partir d'une solution courante $\mathcal{S}$, à explorer partiellement l'espace Ω grâce à des transformations locales dans le but d'améliorer $\mathcal{S}$. Il n'existe pas d'algorithme officiel pour la recherche locale, on peut utiliser diverses techniques pour la mettre en œuvre, généralement ce sont des techniques simples et rapides. On utilise souvent la descente : on effectue des transformations locales successives et seules les transformations améliorantes (qui diminuent le coût de la solution, dans le cas d'un problème de minimisation) sont conservées (voir algorithme 2). Un tel algorithme peut être vu comme induisant une trajectoire de la solution courante dans Ω. Une autre méthode simple et rapide et induisant une trajectoire différente dans Ω est la marche aléatoire (*random walk*). Dans

cet algorithme, on accepte toutes les transformations locales, la solution pouvant donc être dégradée (voir algorithme 3). Plusieurs critères d'arrêt sont envisageables : nombre maximal d'itérations, nombre maximal d'itérations sans amélioration, seuil critique pour le coût de la solution courante ... La figure 1.5 illustre les trajets typiques engendrés par ces deux algorithmes dans l'espace des solutions.

Algorithme 2 : Descente

Sorties : $\mathcal{S}^*$ //meilleure solution trouvée

1 Initialiser une solution $\mathcal{S}$;
2 $Stop \leftarrow faux$;
3 **Tant que** $Stop = faux$ **faire**
4   Calculer $\mathcal{S}'$ le meilleur voisin de $\mathcal{S}$;
5   **Si** $f(\mathcal{S}') \leq f(\mathcal{S})$ **alors**
6         $\mathcal{S} \leftarrow \mathcal{S}'$
7   **sinon**
8         $Stop \leftarrow vrai$;
9 $\mathcal{S}^* \leftarrow \mathcal{S}$;

Algorithme 3 : Marche aléatoire

Sorties : $\mathcal{S}^*$ //meilleure solution trouvée

1 Initialiser une solution $\mathcal{S}$;
2 $f(\mathcal{S}^*) = +\infty$;
3 **Tant que** *critère d'arrêt non atteint* **faire**
4   Choisir $\mathcal{S}'$ dans $\mathcal{V}(\mathcal{S})$;
5   **Si** $f(\mathcal{S}') < f(\mathcal{S}^*)$ **alors**
6         $\mathcal{S}^* \leftarrow \mathcal{S}'$
7     $\mathcal{S} \leftarrow \mathcal{S}'$;

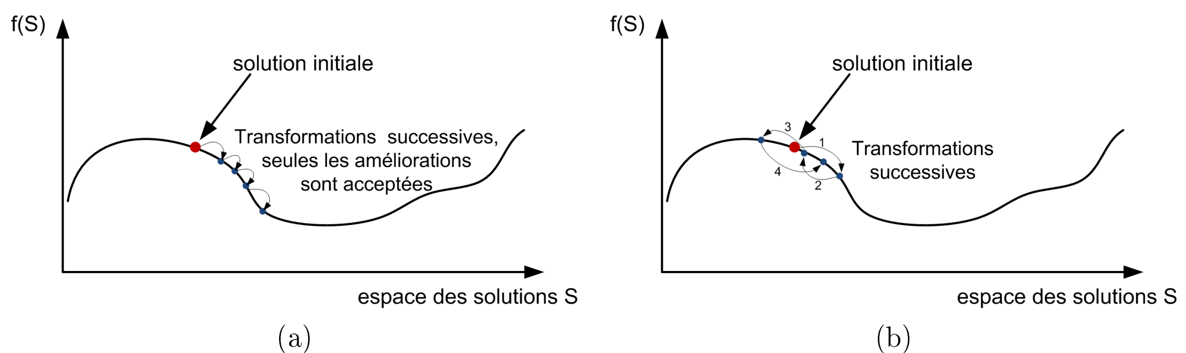


Fig. 1.5 – Comparaison des trajectoires engendrées par la descente (a) et par la marche aléatoire (b)

Exemple 3. *Exemple de descente appliquée au voyageur de commerce.*

Pour illustrer le fonctionnement de la descente, nous allons l'appliquer au voyageur de commerce. Nous reprenons l'instance du tableau 1.1 qui a déjà servi pour illustrer

l'algorithme glouton. Il nous faut également choisir un opérateur de transformation locale pour le calcul des voisins, nous choisissons l'opérateur d'échange pour sa simplicité. Pour bien voir le déroulement de l'algorithme, nous partons d'une mauvaise solution initiale par exemple $\mathcal{S}$ =(0, 3, 1, 4, 2, 0) de coût 365. L'algorithme se déroule alors comme suit :

itération 1 $\mathcal{V}(\mathcal{S}) = \{$ (0, 1, 3, 4, 2, 0), (0, 4, 1, 3, 2, 0), (0, 2, 1, 4, 3, 0), (0, 3, 4, 1, 2, 0), (0, 3, 2, 4, 1, 0), (0, 3, 1, 2, 4, 0)$\}$. Les coûts respectifs des solutions sont 329, 375, 325, 325, 346 et 340. On choisit donc une des deux instances de coût 325, on remarque qu'elles sont symétriques donc ici ce choix n'a pas d'importance. Choisissons par exemple (0, 2, 1, 4, 3, 0). Comme cette solution améliore le coût (on passe de 365 à 325), elle devient la nouvelle solution courante $\mathcal{S}$ et l'algorithme continue.

itération 2 $\mathcal{V}(\mathcal{S}) = \{$(0, 1, 2, 4, 3, 0), (0, 4, 1, 2, 3, 0), (0, 3, 1, 4, 2, 0), (0, 2, 4, 1, 3, 0), (0, 2, 3, 4, 1, 0), (0, 2, 1, 3, 4, 0) $\}$. Les coûts respectifs des solutions sont 300, 346, 365, 365, 335, 329. Le meilleur voisin de $\mathcal{S}$ est donc (0, 1, 2, 4, 3, 0) de coût 300. Comme cette solution améliore le coût (on passe de 325 à 300), elle devient la nouvelle solution courante $\mathcal{S}$ et l'algorithme continue.

itération 3 $\mathcal{V}(\mathcal{S}) = \{$(0, 2, 1, 4, 3, 0), (0, 4, 2, 1, 3, 0), (0, 3, 2, 4, 1, 0), (0, 1, 4, 2, 3, 0), (0, 1, 3, 4, 2, 0), (0, 1, 2, 3, 4, 0) $\}$. Les coûts respectifs des solutions sont 325, 340, 346, 346, 329 et 310 . Le meilleur voisin de $\mathcal{S}$ est donc (0, 1, 2, 3, 4, 0) de coût 310. Comme cette solution n'améliore pas le coût, l'algorithme s'arrête et renvoie la meilleure solution trouvée, c'est-à-dire (0, 1, 2, 4, 3, 0) de coût 300. Sur cet exemple très simple, on a trouvé la solution optimale, cependant, notons que ce n'est pas le cas en général. Les différentes étapes de l'algorithme sont illustrées figure 1.6.

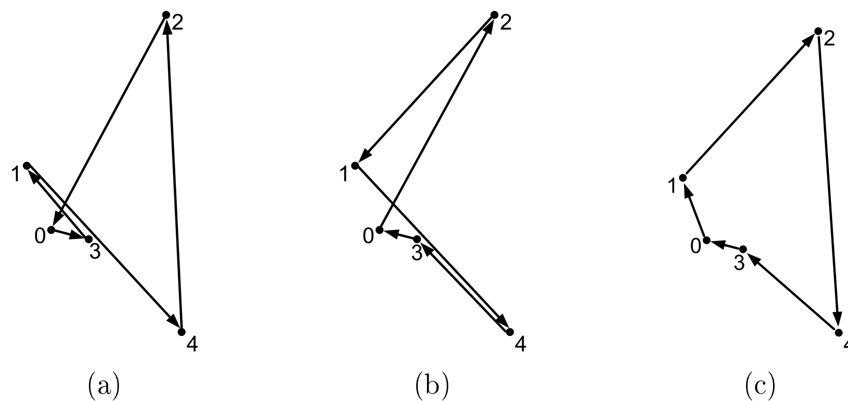


FIG. 1.6 – Tournées calculées par l'algorithme de descente : (a) tournée initiale ; (b) tournée retenue à l'itération 1 (3 et 2 échangés par rapport à la tournée initiale) ; (c) tournée retenue à l'itération 2 (2 et 1 échangés par rapport à la tournée précédente)

La descente et la marche aléatoire peuvent être vues comme des métaheuristiques basiques puisqu'elles proposent une stratégie de recherche dans l'espace des solutions et peuvent s'appliquer à n'importe quel problème d'optimisation combinatoire, pourvu qu'on possède un opérateur de transformation locale. Cependant, elles présentent des défauts majeurs. En effet, la descente qui n'accepte que des transformations améliorantes se dirige rapidement vers des solutions de meilleur coût mais se retrouve aussi rapidement piégée dans un minimum local sans moyen d'en sortir. Notons, ligne 5, l'utilisation du signe $\leq$:

le meilleur voisin $\mathcal{S}'$ de la solution courante $\mathcal{S}$ est accepté même s'il n'est pas strictement meilleur, ceci permet de ne pas rester bloqué sur un palier, cependant, il faut prendre garde à ne pas boucler sur les mêmes solutions. La marche aléatoire permet des transformations qui dégradent le coût de la solution courante, ce qui peut permettre de sortir des minima locaux. Cependant, elle ne présente aucun mécanisme de contrôle et accepte systématiquement toutes les transformations. Le risque est alors de se retrouver dans des régions de l'espace des solutions très peu satisfaisantes ou même de boucler en appliquant une transformation qui annule la précédente. Cet algorithme donne, en général, des résultats médiocres.

Les métaheuristiques que nous présentons dans les sections suivantes proposent des stratégies de recherche plus élaborées. Certaines de ces stratégies ont été conçues par analogie avec d'autres domaines, comme par exemple la physique (recuit simulé) ou la biologie (algorithmes évolutionnaires, colonies de fourmis). La plupart des métaheuristiques présentent l'inconvénient d'avoir de nombreux paramètres à régler dont dépend le compromis délicat entre efficacité et temps de calcul. Nous présentons les métaheuristiques les plus connues, à partir desquelles de nombreuses variantes peuvent être élaborées. Dans les sections suivantes, le terme recherche locale désignera la descente.

1.2.4 Le recuit simulé (*Simulated Annealing* - SA)

Le recuit simulé est une des métaheuristiques les plus connues (développée simultanément par Kirkpatrick *et al.* [KGV83] et Cerny [Cer85]) incluant l'acceptation de transformations dégradant le coût de la solution courante. Elle tire son nom du domaine de la métallurgie (voir [VR09]). Cette métaheuristique est caractérisée par la présence d'une variable de contrôle appelée température (par analogie aux processus thermodynamiques dont elle s'inspire) qui fixe les conditions dans lesquelles une transformation dégradante est acceptée.

Algorithme 4 : Schéma algorithmique du recuit simulé

```
   Entrées : θ0 //meilleure solution trouvée
   Sorties : S*
1  Initialiser une solution S ;
2  θ ← θ0 ;
3  Tant que critère d'arrêt non atteint faire
4  |  Choisir S' voisin de S ;
5  |  Si f(S') < f(S) alors
6  |  |  S ← S' ;
7  |  sinon
8  |  |  Si exp(−(f(S') − f(S))/θ) ≥ random //random : nombre aléatoire entre 0 et 1
   |  |  alors
9  |  |  |  S ← S' ;
10 |  Faire décroître θ ;
11 S* ← la meilleure solution trouvée ;
```

Elle fonctionne comme suit. Initialement on fixe la température θ à une température donnée θ_0 et on génère une solution $\mathcal{S}$ à l'aide d'une heuristique quelconque ; $\mathcal{S}$ devient la solution courante. A chaque itération on choisit une solution $\mathcal{S}'$ dans un voisinage de $\mathcal{S}$, si $\mathcal{S}'$ est meilleure que $\mathcal{S}$ alors $\mathcal{S}'$ devient la solution courante sinon $\mathcal{S}'$ devient la solution courante avec une probabilité dépendant de θ et de la différence de coût entre $\mathcal{S}$ et $\mathcal{S}'$ (voir algorithme 4). Plus la température est haute, plus la probabilité d'accepter une transformation qui dégrade la solution courante est élevée. Au fur et à mesure des itérations, la température diminue de sorte qu'il devient de moins en moins probable d'accepter une solution dégradante. L'efficacité de cet algorithme dépend bien sûr, entre autre, de la stratégie adoptée pour faire décroître θ. [AZ04] propose différents mécanismes de contrôle.

1.2.5 La recherche tabou (*Tabu Search* - TS)

La recherche tabou est également une des métaheuristiques les plus connues, elle a été introduite par Glover [Glo86]. Elle utilise un historique de manière à interdire à l'algorithme de revenir sur ses pas. Cet historique se traduit par la présence d'une liste dite tabou qui garde une trace des dernières solutions visitées, ainsi l'algorithme ne pourra plus explorer ces solutions (du moins à court terme, tout dépend de la taille de la liste tabou). L'algorithme fonctionne comme suit. Initialement la liste tabou est vide et on génère une solution $\mathcal{S}$ à l'aide d'une heuristique quelconque ; $\mathcal{S}$ devient la solution courante. A chaque itération, on choisi le meilleur voisin $\mathcal{S}'$ de $\mathcal{S}$ qui n'est pas déjà dans la liste tabou, $\mathcal{S}'$ devient la solution courante $\mathcal{S}$ et est ajouté à la liste tabou (voir algorithme 5). Si la taille de la liste tabou dépasse la taille maximale autorisée, on supprime de cette liste l'élément le plus ancien (stratégie FIFO - First In First Out). Notons que la liste tabou permet d'éviter les cycles en interdisant de choisir une solution dans le voisinage de la solution courante qui aurait déjà était explorée. En outre, en pratique, on ne stocke pas les solutions dans leur intégralité (trop coûteux en temps et en espace) mais seulement une signature de ces solutions.

Algorithme 5 : Schéma algorithmique de la recherche tabou

Sorties : $\mathcal{S}^*$

1 Initialiser une solution $\mathcal{S}$;
2 $\mathcal{L}_{tabou} \leftarrow \emptyset$;
3 **Tant que** *critère d'arrêt non atteint* **faire**
4   Choisir $\mathcal{S}'$ le meilleur voisin de $\mathcal{S}$ qui n'est pas dans $\mathcal{L}_{tabou}$;
5   Insérer $\mathcal{S}'$ dans $\mathcal{L}_{tabou}$ suivant une stratégie FIFO ;
6     $\mathcal{S} \leftarrow \mathcal{S}'$;
7 $\mathcal{S}^* \leftarrow$ la meilleure solution trouvée ;

1.2.6 La recherche à voisinage variable (*Variable Neighborhood Search* - VNS) et ses dérivés

Comme son nom l'indique, cette métaheuristique repose sur l'utilisation de plusieurs systèmes de voisinages (voir [HM01]). L'idée principale est d'utiliser ces différents voisinages pour sortir des minima locaux. En effet, des voisinages différents induisent une

topologie différente de l'espace des solutions, ainsi, un minimum local d'un point de vue d'un voisinage donné n'est pas forcément un minimum local pour un autre voisinage (voir figure 1.7). Une solution courante peut donc appartenir à différents bassins d'attraction suivant le voisinage considéré. Par ailleurs, l'utilisation de plusieurs transformations locales permet de diversifier les solutions et ainsi de parcourir des régions différentes de l'espace des solutions.

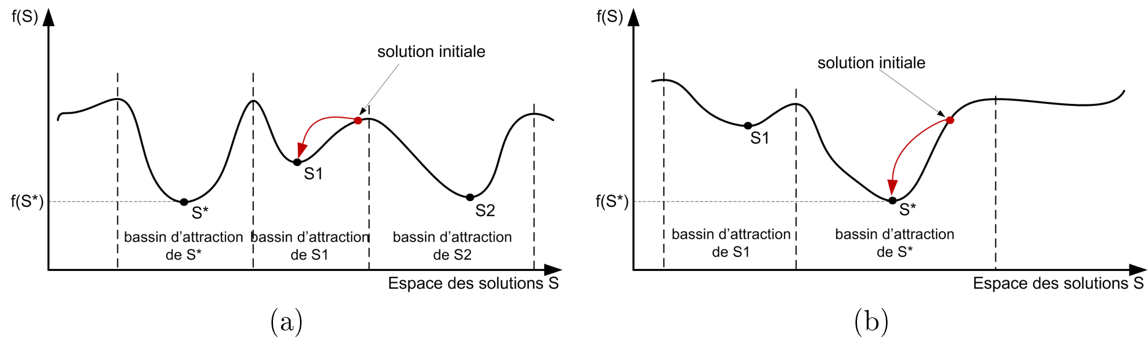


FIG. 1.7 – Deux voisinages différents permettent d'explorer des bassins d'attraction différents : (a) une recherche locale avec le voisinage 1 mène à un optimum local ; (b) une recherche locale avec le voisinage 2 à partir de la même solution initiale mène à l'optimum global

Algorithme 6 : Schéma algorithmique de la recherche à voisinage variable (*Variable Neighborhood Search* - VNS)

Sorties : $\mathcal{S}^*$

1 Initialiser une solution $\mathcal{S}$;
2 **Tant que** *critère d'arrêt non atteint* **faire**
3 $\quad \mathcal{V}_{cour} \leftarrow \mathcal{V}_1$;
4 $\quad$ **Tant que** *tous les voisinages n'ont pas été utilisés* **faire**
5 $\quad\quad$ Choisir aléatoirement $\mathcal{S}'$ dans le voisinage $\mathcal{V}_{cour}$ de $\mathcal{S}$;
6 $\quad\quad$ Améliorer $\mathcal{S}'$ par recherche locale ;
7 $\quad\quad$ **Si** $f(\mathcal{S}') < f(\mathcal{S})$ **alors**
8 $\quad\quad\quad \mathcal{S} \leftarrow \mathcal{S}'$;
9 $\quad\quad\quad \mathcal{V}_{cour} \leftarrow \mathcal{V}_1$;
10 $\quad\quad$ **sinon**
11 $\quad\quad\quad \mathcal{V}_{cour} \leftarrow \mathcal{V}_{cour+1}$;
12 $\mathcal{S}^* \leftarrow$ la meilleure solution trouvée ;

Le schéma de fonctionnement de cette métaheuristique est très général et peut facilement être adapté à divers voisinages. Il fonctionne comme suit. On dispose de n voisinages $(\mathcal{V}_1 \ldots \mathcal{V}_n)$. Initialement on génère une solution $\mathcal{S}$ à l'aide d'une heuristique quelconque ; $\mathcal{S}$ devient la solution courante et le voisinage courant $\mathcal{V}_{cour}$ est fixé à $\mathcal{V}_1$. A chaque itération on choisit un voisin $\mathcal{S}'$ de $\mathcal{S}$ dans $\mathcal{V}_{cour}$ et on l'améliore par recherche locale. Si après la recherche locale $\mathcal{S}'$ est meilleur que $\mathcal{S}$ alors $\mathcal{S}'$ devient la solution courante $\mathcal{S}$ et le voisinage courant $\mathcal{V}_{cour}$ est réinitialisé à $\mathcal{V}_1$. Dans le cas contraire, on change de voisinage : $\mathcal{V}_{cour}$

devient $\mathcal{V}_{cour+1}$ et on recommence l'itération. Ce processus peut être répété plusieurs fois (voir algorithme 6).

Il existe des variantes de cette métaheuristique comme la descente à voisinage variable (*Variable Neighborhood Descent* - VND) qui choisit le meilleur voisin $\mathcal{S}'$ dans le voisinage de $\mathcal{S}$ (au lieu de choisir aléatoirement) ; ou encore la recherche décomposée à voisinage variable (*Variable Neighborhood Decomposition Search* - VNDS) qui utilise des voisinages qui agissent, non pas sur l'ensemble de la solution courante, mais seulement sur certains attributs de cette solution, chaque voisinage agissant sur des attributs différents (voir [BR03]). Ces trois métaheuristiques s'échappent des minima locaux en utilisant différents voisinages. Dans le même esprit, la recherche locale guidée (*Guided Local Search* - GLS) utilise, non pas différents voisinages, mais différentes fonctions objectifs de sorte que les minima locaux du point de vue d'une fonction f soient différents du point de vue d'une autre fonction f'.

1.2.7 L'algorithme GRASP (*Greedy Randomized Adaptative Search Procedure*)

L'algorithme GRASP (introduit par Feo et Resende [FR89]) est une métaheuristique à démarrages multiples . Elle consiste à initialiser une solution grâce à un algorithme glouton randomisé puis à l'améliorer par une recherche locale. On répète itérativement ces deux phases et on conserve la meilleure solution trouvée (voir algorithme 7). Le GRASP permet de quadriller l'espace de recherche.

Algorithme 7 : Schéma algorithmique GRASP

Entrées : $iterMax$
Sorties : $\mathcal{S}^*$

1 **Pour** $i = 1$ *à* $iterMax$ **faire**
2   Initialiser une solution $\mathcal{S}$ grâce à un algorithme glouton ;
3   Améliorer $\mathcal{S}$ grâce à une recherche locale ;
4 $\mathcal{S}^* \leftarrow$ la meilleure solution $\mathcal{S}$ trouvée ;

1.2.8 La recherche locale itérée (*Iterated Local Search* - ILS)

La recherche locale itérée est une métaheuristique qui utilise à la fois mutation et recherche locale (voir [LMS03]). Elle fonctionne suivant le schéma suivant. Initialement une solution est générée (aléatoirement ou grâce à une heuristique gloutonne) et améliorée par recherche locale. Elle devient alors la solution courante $\mathcal{S}$. La solution courante $\mathcal{S}$ est perturbée par un opérateur de mutation et devient $\mathcal{S}'$. Cette nouvelle solution est améliorée par une recherche locale. Si $\mathcal{S}'$ vérifie un critère donné (critère d'acceptation) alors $\mathcal{S}'$ devient la nouvelle solution courante (dans le cas contraire on conserve $\mathcal{S}$ comme solution courante). Ce schéma est répété itérativement (voir algorithme 8). Notons que certains auteurs (par exemple Prins [Pri09a]) donne une définition de l'ILS dans laquelle on accepte la nouvelle solution $\mathcal{S}'$ uniquement dans le cas où elle est meilleure que la solution courante $\mathcal{S}$.

Algorithme 8 : Schéma algorithmique ILS

Entrées : $iterMax$
Sorties : $\mathcal{S}^*$

1 Initialiser une solution $\mathcal{S}$;
2 Améliorer $\mathcal{S}$ grâce à une recherche locale ;
3 **Pour** i = *1 à* $iterMax$ **faire**
4 Transformer $\mathcal{S}$ en $\mathcal{S}'$ grâce à un opérateur de mutation ;
5 Améliorer $\mathcal{S}'$ grâce à une recherche locale ;
6 **Si** $\mathcal{S}'$ *vérifie le critère d'acceptation* **alors**
7 $\mathcal{S} \leftarrow \mathcal{S}'$;
8 $\mathcal{S}^* \leftarrow$ la meilleure solution $\mathcal{S}'$ trouvée ;

L'opérateur de mutation a un rôle prépondérant dans l'efficacité de cette métaheuristique : s'il induit des perturbations trop petites les solutions manquent de diversité, la recherche locale risque alors de mener toujours au même optimum local, dans le cas contraire (les perturbations sont trop importantes) $\mathcal{S}$ et $\mathcal{S}'$ sont totalement différentes et l'algorithme se comporte alors comme un GRASP. L'idéal est d'avoir un opérateur de mutation qui génère une nouvelle solution dans un bassin d'attraction adjacent à la solution courante. Le critère d'acceptation tient également un rôle important dans le comportement de l'algorithme. Il permet d'ajuster le niveau d'intensification et de diversification. Imaginons un critère qui n'accepte la nouvelle solution $\mathcal{S}'$ que si elle est meilleure que la solution courante $\mathcal{S}$, on favorise alors grandement l'intensification. Au contraire, si on considère un critère qui accepte systématiquement la nouvelle solution $\mathcal{S}'$, on favorise la diversification.

1.2.9 La recherche locale évolutionnaire(*Evolutionary Local Search*-ELS)

La recherche locale évolutionnaire, récemment introduite par [WM07], est une forme évoluée de l'ILS.

Algorithme 9 : Schéma algorithmique ELS

Entrées : $iterMax$, $nbFils$
Sorties : $\mathcal{S}^*$

1 Initialiser une solution $\mathcal{S}$;
2 Améliorer $\mathcal{S}$ grâce à une recherche locale ;
3 **Pour** i = *1 à* $iterMax$ **faire**
4 $f(\mathcal{S}'') = +\infty$;
5 **Pour** j = *1 à* $nbFils$ **faire**
6 Transformer $\mathcal{S}$ en $\mathcal{S}'$ grâce à un opérateur de mutation ;
7 Améliorer $\mathcal{S}'$ grâce à une recherche locale ;
8 **Si** $f(\mathcal{S}') < f(\mathcal{S}'')$ **alors**
9 $\mathcal{S}'' \leftarrow \mathcal{S}'$;
10 **Si** $\mathcal{S}''$ *vérifie le critère d'acceptation* **alors**
11 $\mathcal{S} \leftarrow \mathcal{S}''$;
12 $\mathcal{S}^* \leftarrow$ la meilleure solution $\mathcal{S}''$ trouvée ;

La différence entre l'ELS et l'ILS vient du fait qu'elle utilise, à chaque itération, n fois l'opérateur de mutation afin de générer n fils de la solution courante (au lieu d'un seul comme le fait ILS), voir algorithme 9 et schéma 1.8. De cette manière, on couvre mieux les bassins d'attraction adjacents à la solution courante.

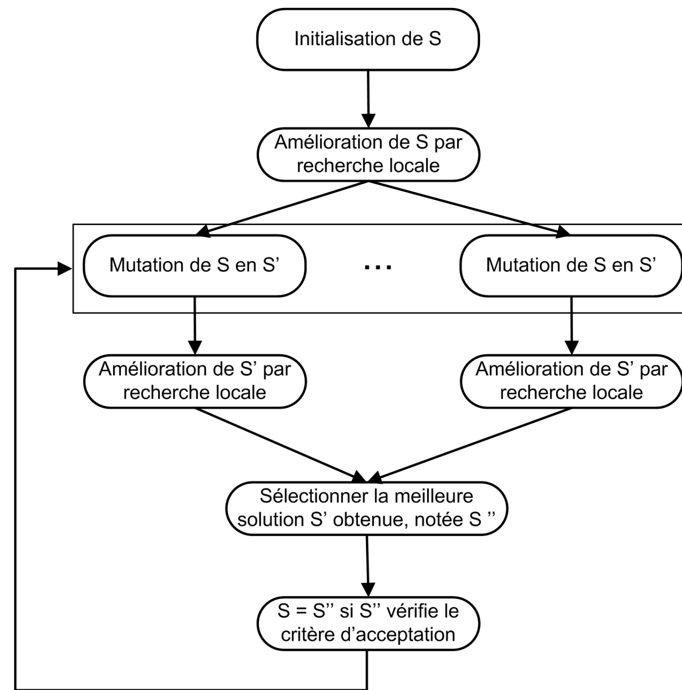


Fig. 1.8 – Schéma représentant l'enchaînement des étapes d'une ELS

1.2.10 Hybridation GRASP×ILS et GRASP×ELS

Dans [Pri09a], Prins propose une hybridation entre GRASP et ILS et entre GRASP et ELS (que l'on notera respectivement GRASP×ILS et GRASP×ELS). L'idée est de tirer profit des points forts de chaque méthode. On remplace la recherche locale du GRASP par une ILS ou une ELS (le schéma 1.9(c) présente le GRASP×ILS). Cette méthode apporte une grande diversité dans les solutions grâce au GRASP et intensifie la recherche locale grâce au schéma ILS ou ELS.

Les métaheuristiques que nous venons de présenter (recuit simulé, recherche tabou, VNS, GRASP, ILS, ELS, GRASP×ILS, GRASP×ELS) ont la particularité de ne manipuler qu'une seule solution au cours du processus de transformation. Elles induisent une trajectoire de la solution courante dans l'espace de recherche. C'est pourquoi, on leur donne parfois le nom de « méthodes de trajectoire ». D'autres métaheuristiques manipulent une famille ou population de solutions à chaque itération, c'est le cas des algorithmes évolutionnaires et des algorithmes de colonie de fourmis que nous présentons dans les sections suivantes.

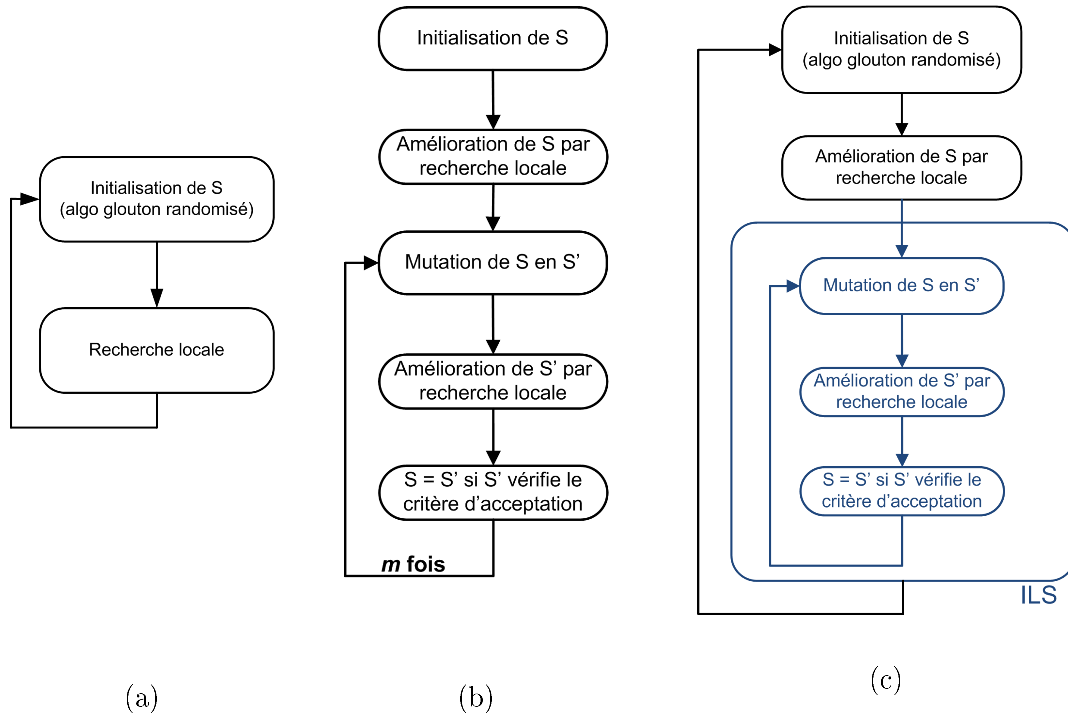


FIG. 1.9 – (a) GRASP ; (b) ILS ; (c) GRASP×ILS

1.2.11 Les algorithmes évolutionnaires et mémétiques

Les algorithmes évolutionnaires sont inspirés du domaine de la biologie : ils ont été introduits dans les années 1950 ([Fra57]). Les techniques utilisées reproduisent le schéma d'évolution des espèces et en adoptent le vocabulaire. Les solutions sont appelées individus et l'algorithme traite simultanément plusieurs individus. L'ensemble de ces individus est appelé population et évolue à chaque itération de l'algorithme. La population relative à une itération donnée s'appelle génération, on obtient donc une nouvelle génération à chaque itération. Les individus qui servent à produire la nouvelle génération sont appelés parents et les individus résultants sont appelés enfants ou fils. Le principe des algorithmes évolutionnaires est simple : l'idée est de faire évoluer une population initiale grâce à des mécanismes de reproduction et de sélection de manière à obtenir des individus de plus en plus performants. Dans le cadre des problèmes de minimisation un individu $\mathcal{S}$ est d'autant plus performant que son coût $(f(\mathcal{S}))$ est faible.

Le schéma général de cet algorithme fonctionne comme suit. Initialement, on génère aléatoirement n individus, ils constituent la population initiale. A chaque itération on choisit des individus pour la reproduction et on leur applique des opérateurs de croisement et de mutation. Les opérateurs de croisement (souvent désignés par leur équivalent anglais *crossover*) produisent un ou deux enfants à partir de deux parents tandis que les opérateurs de mutation produisent un enfant à partir d'un unique individu. La performance des nouveaux individus est évaluée et un opérateur de sélection choisit les individus qui appartiendront à la prochaine génération (nouvelle population courante). Le mécanisme de sélection permet de garder une population de taille constante et favorise la survie des individus les plus per-

formants. L'algorithme 10 illustre le mécanisme général d'un algorithme évolutionnaire. La fonction `Selection_reproduction`(P) choisit dans P les individus pour la reproduction. La fonction `Croisement`(P) applique des opérateurs de croisement sur des individus de P ; la fonction `Mutation`(P) applique des opérateurs de mutation sur des individus de P ; enfin la fonction `Selection`(P) sélectionne les individus de P qui appartiendront à la prochaine génération.

Algorithme 10 : Schéma algorithmique d'un algorithme évolutionnaire

Sorties : $\mathcal{S}^*$

1. Générer une population initiale P ;
2. Evaluer la performance des individus de P ;
3. **Tant que** *critère de fin non atteint* **faire**
4. $\quad P_{repro} \leftarrow$ **Selection_reproduction** (P) ;
5. $\quad P' \leftarrow$ **Croisement** (P_{repro}) ;
6. $\quad P' \leftarrow$ **Mutation** (P') ;
7. $\quad$ Evaluer la performance des individus de P' ;
8. $\quad P \leftarrow$ **Selection** ($P \cup P'$) ;
9. $\mathcal{S}^* \leftarrow$ le meilleur individu trouvé ;

Plusieurs approches d'algorithmes évolutionnaires ont été étudiées (voir [SJB+93] pour le détail de ces trois approches) :

- les stratégies d'évolution (*evolution strategies* - ES), utilisées pour résoudre des problèmes d'optimisation continue ;
- la programmation évolutionnaire (*evolutionary programming* - EP), conçue pour faire évoluer des automates à états finis, l'idée étant de créer une intelligence artificielle ;
- les algorithmes génétiques (*genetic algorithms* - GA), utilisés pour résoudre des problèmes d'optimisation combinatoire, ces derniers étant certainement les plus populaires.

A l'origine, ces trois approches ont été développées en parallèle s'ignorant mutuellement. Aujourd'hui les algorithmes évolutionnaires sont massivement utilisés, ce qui conduit à des techniques de plus en plus sophistiquées et à de nombreuses variantes. Pour plus d'information sur ces techniques on peut se référer à [Ash06] en particulier aux chapitres 1 et 2 qui présentent de manière générale les algorithmes évolutionnaires puis en détaillent les différentes caractéristiques ainsi que la manière de les implémenter (les chapitres suivants sont plus spécifiques et s'intéressent chacun à une problématique spécifique).

Les algorithmes évolutionnaires manipulent une population de solutions qu'ils font évoluer grâce des mécanismes basés uniquement sur des opérateurs de croisement, mutation et sélection. Au cours des années, il est devenu classique d'ajouter une recherche locale avant la phase de sélection des nouveaux individus (ce qui rend l'algorithme bien plus efficace) créant ainsi un algorithme hybride « algorithme évolutionnaire / recherche locale ». Aujourd'hui ces algorithmes hybrides sont connus sous le nom d'algorithmes mémétiques.

1.2.12 Scatter Search et Path Relinking

L'idée originale de la recherche dispersée, plus connue sous le nom anglais *scatter search*, a été introduite par Glover dans [Glo77] en 1977. Le *scatter search* est une approche évolutionnaire, elle opère donc sur une population de solutions et possède des procédures pour combiner ces solutions et en créer de nouvelles. Le *scatter search* diffère des algorithmes évolutionnaires classiques évoqués en section 1.2.11 en deux points principaux :

1. elle n'utilise pas de procédés aléatoires pour maintenir la diversité des solutions ;
2. elle utilise une stratégie de génération de nouvelles solutions par combinaisons linéaires (et non par croisements et mutations).

Notons que, même si dans la définition initiale du *scatter search* il n'existe pas de composante aléatoire, en pratique de nombreux auteurs en introduisent pour gérer la diversité des solutions (voir par exemple [CLP06]).

Algorithme 11 : Algorithmique de principe du *Scatter Search*

Sorties : $\mathcal{S}^*$

```
1  Tant que nombre d'itération maximal non atteint faire
2  |   Générer un ensemble T diversifié de solutions ;
3  |   Améliorer les solutions de T par un procédé heuristique ;
4  |   Choisir un sous-ensemble T_ref de T comme ensemble de référence ;
5  |   Stop ← faux ;
6  |   Tant que !Stop faire
7  |   |   Créer des nouvelles solutions grâce à une combinaison linéaire de solutions
   |   |   de T_ref ;
8  |   |   Réparer si besoin les solutions obtenues pour les rendre « acceptables » ;
9  |   |   Améliorer ces solutions par un procédé heuristique ;
10 |   |   Choisir les meilleures solutions générées et les ajouter à T_ref ;
11 |   |   Si T_ref est inchangé alors
12 |   |   |   Stop ← vrai ;
13 S* ← le meilleur individu trouvé au cours du processus ;
```

L'algorithme de principe est donné par l'algorithme 11. Initialement, un ensemble de solutions est généré (on s'assure d'un niveau minimal de diversité) puis les solutions sont améliorées par un procédé heuristique spécifique au problème considéré. Les meilleures solutions de l'ensemble ainsi créées sont choisies pour former l'ensemble de référence T_{ref}. Notons qu'ici la notion de « meilleure solution » n'est pas limitée à la mesure donnée par la fonction objectif, une solution peut être incluse dans l'ensemble de référence pour la diversité qu'elle génère. Cet ensemble de référence est ensuite amélioré itérativement. Des nouvelles solutions sont créées par combinaisons linéaires de solutions de T_{ref} puis réparées, si besoin, pour les rendre « acceptables » (par exemple si on manipule des vecteurs d'entiers la combinaison linéaire peut engendrer des vecteurs de réels, le mécanisme de réparation consiste alors à rendre les nouveaux vecteurs entiers). Notons qu'une solution acceptable n'est pas forcément réalisable. Les nouvelles solutions sont ensuite améliorées par le même procédé que dans la phase initiale, puis les meilleures d'entre elles sont incluses dans l'ensemble de référence. On répète la phase d'amélioration de l'ensemble T_{ref}

jusqu'à ce qu'il ne change plus. Pour ajouter de la diversité, on génère un nouvel ensemble de référence, recommençant ainsi le processus depuis sa phase initiale. On arrête lorsqu'on a atteint un nombre donné d'itérations.

Notons que, d'un point de vue géométrique, la génération de combinaisons linéaires entre deux solutions induit un chemin entre ces solutions. La stratégie qui consiste alors à utiliser ces chemins pour créer de nouvelles solutions constitue le fondement de la métaheuristique appelée *path relinking*. *path relinking* est une extension du *scatter search* qui s'appuie sur la recherche de chemins. Le lecteur peut trouver plus de détails concernant ces deux métaheuristiques dans [GLM00].

Notons que l'hybridation du *path relinking* et du GRASP a déjà été testée et donne de très bons résultats (voir [LM99] par exemple).

1.2.13 L'optimisation par colonies de fourmis (*Ant Colony Optimization - ACO)*

Les algorithmes de colonies de fourmis s'inspirent du comportement de certains insectes sociaux, en particulier des fourmis (cette approche est due à Colorni et al. [CDM92]). Ces dernières parviennent collectivement à résoudre des problèmes trop complexes pour un unique individu. Pour bien comprendre le mécanisme de ces algorithmes, voyons comment les fourmis résolvent le problème qui consiste à trouver le plus court chemin entre leur nid et une source de nourriture. Lorsqu'elles sont à la recherche de nourriture, les fourmis explorent l'espace autour de leur nid de manière aléatoire. En se déplaçant, elles laissent des phéromones sur le sol. Lorsqu'une fourmi trouve une source de nourriture, elle retourne à son nid. Les fourmis qui ont emprunté le plus court chemin arrive plus tôt à la source de nourriture, elles reprennent donc le même chemin pour le retour avec une forte probabilité renforçant ainsi la présence de phéromones sur ce trajet. Une fourmi qui cherche son chemin aura tendance à suivre une piste déjà marquée par des phéromones. La plus forte présence de phéromones sur le plus court chemin encourage les autres fourmis à suivre ce trajet pour se rendre à la source de nourriture. A force d'allers-retours, ce chemin est de plus en plus marqué et sera donc suivi à terme par l'ensemble de la colonie.

Il existe plusieurs schémas algorithmiques pour résoudre un problème d'optimisation par colonies de fourmis. Ces schémas sont assez complexes à formaliser de manière simple et compréhensible sous forme d'algorithme de principe. Cependant, leur point commun est d'inclure un modèle de phéromones qui sert à guider la recherche de solutions vers les régions plus prometteuses de l'espace de recherche. Ils fonctionnent en deux étapes :

étape 1 : des solutions sont choisies dans l'espace de recherche en utilisant un modèle donné de phéromones ;

étape 2 : les solutions générées sont évaluées et leur coût sert à modifier le modèle de phéromones de manière à concentrer la recherche de solutions dans les régions contenant des solutions de bonne qualité.

Pour plus d'information sur ces techniques, le lecteur peut se référer à [Blu05] qui présente, entre autre, le modèle biologique à l'origine des algorithmes de colonies de fourmis,

le schéma général de résolution par colonies de fourmis et les variantes algorithmiques les plus répandues.

1.2.14 Discussion

1.2.14.1 Implémentations parallèles

De nombreuses métaheuristiques sont bien adaptées à une implémentation parallèle. Par exemple, dans le GRASP, les générations de solutions à l'aide de l'algorithme glouton sont indépendantes et peuvent donc être parallélisées. Ceci présente l'avantage de réduire considérablement les temps de calcul en tirant profit des processeurs actuels qui sont tous multi-cœurs.

1.2.14.2 Recours à des pénalités pour les problèmes fortement contraints

Toutes les méthodes présentées dans cette section ont la particularité de garder toujours la solution courante dans l'espace Ω, autrement dit, la solution courante ne viole jamais les contraintes. Cependant, dans le cadre de problèmes fortement contraints, il n'est pas toujours possible de trouver un opérateur de transformation locale induisant un voisinage dans Ω ou même de générer une solution initiale respectant les contraintes *via* un algorithme glouton. Une stratégie couramment utilisée consiste alors à relaxer les contraintes pour revenir ensuite vers une solution faisable. Pour cela, on utilise un schéma de pénalité : le coût associé à une solution $\mathcal{S}$ n'est plus $f(\mathcal{S})$ mais $f'(\mathcal{S}) = f(\mathcal{S}) + p(\mathcal{S})$ où $p(\mathcal{S})$ est la pénalité associée à $\mathcal{S}$. Intuitivement, plus $\mathcal{S}$ viole les contraintes, plus sa pénalité est élevée. On peut alors, avec cette nouvelle fonction coût f', utiliser n'importe laquelle des métaheuristiques décrites ci-dessus. En cherchant à améliorer le coût de la solution les recherches locales ont tendance à minimiser le poids de cette pénalité ramenant ainsi les solutions dans l'espace Ω.

1.2.15 Synthèse sur les métaheuristiques

Comme nous l'avons vu à travers les sections 1.2.4 à 1.2.13, les métaheuristiques offrent des stratégies très diverses de recherche de solutions. Cependant, elles ont en commun d'apporter des solutions à un certain nombre de problèmes en particulier :

les minima locaux : pour sortir d'un miminum local, une métaheuristique doit accepter une dégradation temporaire de la solution en effectuant des mouvements qui augmentent le coût de la solution courante. Pour éviter une divergence du procédé, il est indispensable de mettre en œuvre un mécanisme de contrôle. Cependant, il est difficile de prévoir à quel point la solution doit être dégradée pour sortir du minimum local, ce qui rend le réglage de ce mécanisme très délicat ;

les opérateurs de transformations locales : comme on l'a vu avec la VNS, des transformations locales différentes peuvent changer complètement le voisinage d'une solution, les opérateurs de transformations locales choisis ont donc une importance capitale. Ils peuvent favoriser ou non la convergence de la méthode vers des « bonnes » régions de l'espace des solutions ;

le réglage des paramètres : les métaheuristiques sont paramétrées (toutes ont au moins un « critère de fin » qu'il faut déterminer), le réglage se fait généralement de manière expérimentale même s'il existe des résultats théoriques pour certaines métaheuristiques (par exemple pour l'algorithme du kangourou [Fle93]) ;

la génération de solutions initiales : un dernier point primordial est la génération des solutions initiales, cette étape intervient dans toutes les métaheuristiques. Il est clair que plus les solutions initiales sont de bonne qualité, plus la métaheuristique a de chance de visiter des régions de l'espace des solutions intéressantes.

Le choix d'une métaheuristique est donc délicat. D'une part, il n'existe pas de théorème qui puisse guider l'utilisateur vers une métaheuristique plutôt qu'une autre. D'autre part, l'efficacité d'une métaheuristique dépend du problème considéré et de la façon dont l'utilisateur règle ses différents paramètres.

Afin de donner au lecteur un aperçu général des méthodes les plus utilisées dans la résolution des problèmes d'optimisation combinatoire, nous présentons dans la section suivante deux méthodes exactes très connues et très utilisées : l'exploration arborescente et la programmation linéaire.

1.3 Un aperçu de deux méthodes exactes

Dans cette section, nous présentons deux méthodes exactes : l'exploration arborescente et la programmation linéaire. Nous voyons comment l'exploration arborescente peut être également envisagée de manière heuristique. Leur fonctionnement est illustré sur l'exemple du problème de voyageur de commerce.

1.3.1 L'exploration arborescente

On se place de nouveau dans le cadre de problèmes de minimisation tels que définis dans la section 1.2.1.

Les algorithmes d'exploration arborescente induisent une énumération complète (ou partielle dans certains cas) de l'espace des solutions d'un problème. Une telle énumération se traduit par l'exploration d'un arbre virtuel. La construction de cet arbre induit, comme pour l'algorithme glouton (qui peut être vu comme la génération d'un chemin de l'arbre), la manipulation de solutions partielles. On doit donc être en mesure, pour le problème que l'on traite, de définir l'ensemble des solutions partielles, le coût associé à une solution partielle et la manière dont on ajoute un élément à une solution partielle. L'arbre d'exploration est défini de la manière suivante : sa racine est la solution partielle vide (ou triviale), une feuille est une solution complète au problème (ou un échec si la solution viole les contraintes), un nœud est une solution partielle et les fils d'un nœud N sont toutes les solutions partielles qu'il est possible de construire à partir de la solution partielle associée à N en y ajoutant un unique élément. La taille des solutions partielles augmente donc à chaque niveau de l'arbre de sorte qu'au niveau i de l'arbre on connaît toutes les solutions partielles incluant i éléments de plus que la solution partielle associée à la racine.

Pour illustrer cette technique d'énumération, la figure 1.10 donne l'arbre complet pour le problème du voyageur de commerce qui a déjà servi à illustrer l'algorithme glouton et dont l'instance est donnée par le tableau 1.1. La racine est initialisée avec la tournée triviale (0, 0). A chaque niveau de l'arbre, on énumère toutes les solutions partielles qui ajoutent une nouvelle ville juste avant le retour à la ville 0. Cette énumération est la plus basique que l'on peut faire : elle énumère « bêtement » toutes les solutions possibles de cette instance sans même inclure de considération sur la symétrie du problème.

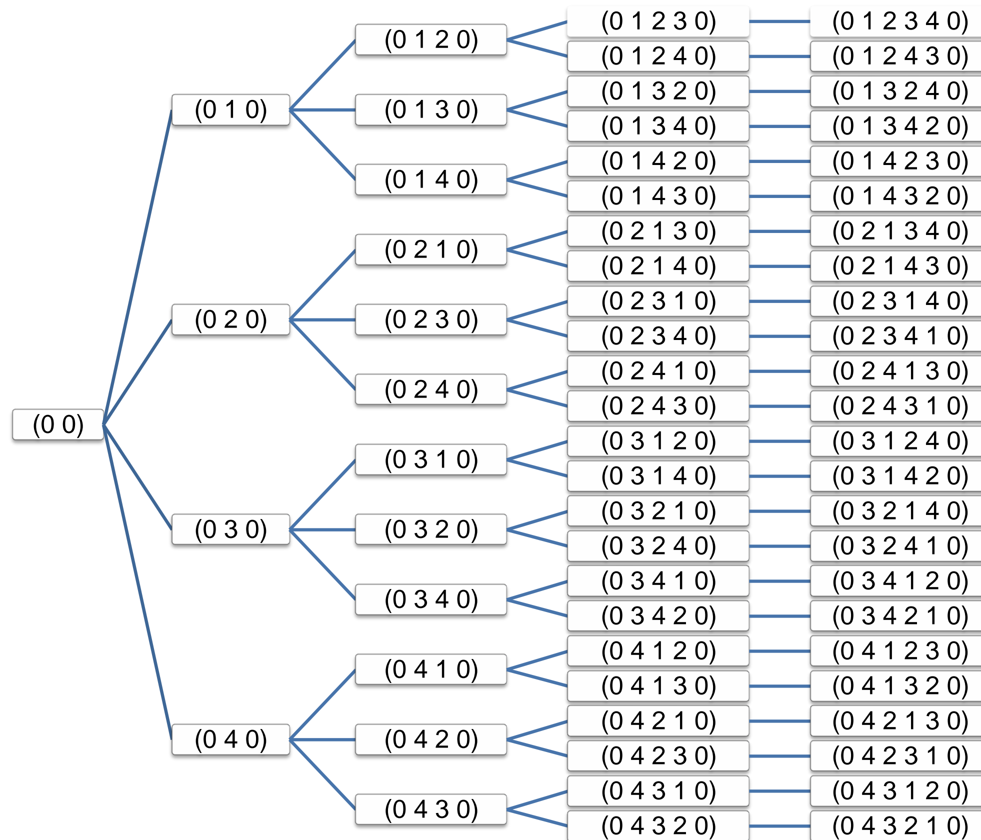


Fig. 1.10 – Enumération de toutes les solutions du TSP à 5 villes à l'aide d'un arbre

On remarque que, contrairement à l'algorithme glouton, on obtient non pas une unique solution à la fin du processus mais toutes les solutions du problème. Il suffit alors de choisir celle de coût le plus faible pour avoir la solution optimale. Ce processus énumératif est bien sûr extrêmement coûteux en temps de calcul étant donné l'explosion combinatoire des solutions. Il ne peut être appliqué qu'à des instances de petite taille. Afin de réduire les temps de calcul, les algorithmes basés sur une exploration arborescente utilisent des techniques de filtrage, c'est-à-dire qu'ils mettent en œuvre des procédés susceptibles d'aider à réduire la taille de l'arbre effectivement exploré. Nous allons voir deux techniques d'exploration arborescente qui incluent des mécanismes de filtrage : un filtrage par séparation-évaluation (*branch & bound*) qui induit une énumération de toutes les solutions potentiellement optimales (on a donc la garantie d'obtenir la solution optimale) et un filtrage heuristique qui se base sur des critères empiriques pour choisir les parties de l'arbre à explorer (on n'a

plus, dans ce cas, la garantie d'obtenir la solution optimale).

1.3.1.1 La méthode par séparation-évaluation (*branch & bound*)

La méthode par séparation-évaluation consiste à générer progressivement tout l'espace de recherche (que l'on représente comme les feuilles d'un arbre comme nous l'avons vu dans la section précédente). Pour gagner du temps, on souhaite ne pas générer toutes les feuilles, leur nombre étant exponentiel par rapport à la taille de l'instance ($(n-1)!$ pour le TSP à n villes ou $\frac{(n-1)!}{2}$ si on tient compte de la symétrie du problème). Le principe est de générer l'arbre progressivement en essayant de commencer par les sous-arbres les plus prometteurs et surtout en n'explorant pas jusqu'aux feuilles un sous-arbre dont on sait qu'il ne peut pas contenir la solution optimale.

Rappelons que nous nous plaçons dans le cadre de problèmes de minimisation, pour des problèmes de maximisation il faut bien sûr changer les comparaisons. Nous décrivons dans cette section le fonctionnement général d'un tel algorithme et l'illustrons sur un exemple. Le lecteur pourra trouver plus de détails concernant les techniques par séparation et évaluation dans le livre [BS05] dédié à la méthode de *branch & bound* appliquée à des problèmes d'analyse de données combinatoires.

1.3.1.1.1 La séparation

Le terme séparation fait référence à la stratégie d'énumération de toutes les solutions, c'est la base de la méthode. Cette stratégie consiste à construire l'arbre d'exploration. Pour cela on a besoin de déterminer :

- un opérateur de construction qui permet d'ajouter un élément à une solution partielle (et donc qui permet de calculer les fils d'un nœud)
- une stratégie de parcours de l'arbre qui détermine l'ordre d'exploration des nœuds.

Au fur et à mesure de l'exploration, on obtient des informations qui permettent, grâce à la phase « évaluation », d'éviter l'exploration de certaines parties de l'arbre.

1.3.1.1.2 L'évaluation

L'évaluation est le point crucial de cette méthode. Elle permet d'exclure, au cours du processus d'exploration, certaines branches de l'arbre dont on sait qu'elles ne peuvent pas contenir la solution optimale. Pour cela on doit disposer des éléments suivants :

- une borne supérieure sur la solution optimale ;
- une fonction d'évaluation d'une solution partielle ;
- une fonction d'estimation du coût de complétion minimal d'une solution partielle (ou une borne inférieure).

La borne supérieure $\mathcal{B}_{sup}$ est généralement initialisée par une heuristique : la solution optimale $\mathcal{S}^*$ étant par définition, la solution de plus faible coût, on a pour toute solution $\mathcal{S}$: $f(\mathcal{S}) \geq f(\mathcal{S}^*)$. Par conséquent, le coût de la solution générée par une heuristique est une

borne supérieure au coût de la solution optimale. Si on ne possède pas d'heuristique pour initialiser la borne supérieure, on peut lui donner la valeur $+\infty$. Au cours du processus d'exploration, si on trouve des solutions $\mathcal{S}$ de coût $f(\mathcal{S})$ inférieur à la borne $\mathcal{B}_{sup}$ alors on affecte $f(\mathcal{S})$ à $\mathcal{B}_{sup}$. De cette manière, la valeur de $\mathcal{B}_{sup}$ décroît au cours de l'algorithme devenant de plus en plus proche du coût de la solution optimale.

La somme du coût d'une solution partielle $\mathcal{S}_p$ et du coût minimal de sa complétion donne une borne inférieure $\mathcal{B}_{inf}$ au coût des solutions complètes qui peuvent être construites à partir de $\mathcal{S}_p$. Si $\mathcal{B}_{inf}$ est supérieur à $\mathcal{B}_{sup}$ (c'est-à-dire supérieur au coût de la meilleure solution trouvée jusqu'alors). Cela indique que les solutions construites à partir de $\mathcal{S}_p$ ne sont pas optimales, il est donc inutile de poursuivre l'exploration dans cette branche. La branche est alors supprimée (on dit parfois « élaguée »). On constate ici l'importance de posséder à la fois une bonne borne $\mathcal{B}_{sup}$ (de coût le plus proche de l'optimal possible) et une bonne fonction d'estimation (qui estime le coût de la solution complète le plus précisément possible tout en restant très rapide) pour élaguer l'arbre un maximum et ainsi gagner en temps de calcul en évitant des explorations inutiles.

1.3.1.1.3 Le parcours de l'arbre

On peut distinguer deux stratégies de parcours :

- Parcours en profondeur : on descend dans les branches jusqu'à trouver une solution (feuille) ou une solution que l'on peut éliminer, dans ce dernier cas, on remonte dans la branche pour redescendre dans une autre direction. On peut choisir de parcourir les branches dans l'« ordre » de l'arbre (de gauche à droite) ou alors en commençant par les nœuds de coût les plus faibles dans un même niveau (qui semblent les plus prometteurs).
- Parcours en largeur : on explore les solutions niveau par niveau. Cette stratégie est rarement utilisée en pratique car elle ne permet pas un filtrage efficace et elle est très coûteuse en place mémoire.

1.3.1.1.4 Exemple de séparation et évaluation appliqué au problème du voyageur de commerce

On reprend le problème du voyageur de commerce qui comprend $n = 5$ villes dont les distances sont données par le tableau 1.1, la ville de départ est la ville 0 :

- la solution partielle triviale est (0, 0), c'est la racine de l'arbre ;
- les solutions partielles sont les tournées $(0, v_1, \ldots, v_i, 0)$ qui visitent seulement $i < n - 1$ villes autre que la ville de départ ;
- l'opérateur de construction insère une nouvelle ville dans la tournée partielle juste avant le retour à la ville de départ ;
- la borne supérieure de la solution optimale est calculée par l'algorithme glouton de la section 1.2.2. Supposons qu'il ait donné une solution de valeur 310 (on ne prend pas la solution qu'on a effectivement calculée de valeur 300 pour ne pas démarrer le

branch and bound avec la valeur de la solution optimale) ;
- la fonction d'évaluation d'une solution partielle calcule le coût de la tournée associée ;
- la fonction d'estimation f_e donne une borne inférieure du coût de complétion d'une solution partielle $\mathcal{S}_p = (0, v_1, \ldots, v_i, 0)$. Elle s'écrit :

$$f_e(\mathcal{S}_p) = -d(v_i, 0) + \min_{j \in \{1 \ldots n-1\}} d(v_i, j) + \min_{j \in \{1 \ldots n-1\}} d(j, 0) + (n-i-2) \times \min_{j,i \in \{1 \ldots n-1\}} d(i, j)$$

où $d(i, j)$ représente la distance de la ville i à la ville j.
En effet, on insère une nouvelle ville entre v_i et 0 donc l'arc $(v_i, 0)$ disparaît, on retranche la distance associée $d(v_i, 0)$; une autre ville est insérée après v_i on ajoute donc un arc de coût minimal $\min_{j \in \{1 \ldots n-1\}} d(v_i, j)$; de même la nouvelle ville avant 0 induit un nouvel arc de coût minimal $\min_{j \in \{1 \ldots n-1\}} d(j, 0)$; enfin la solution complète contiendra $n-i-1$ arcs de plus que la solution partielle du niveau i, $i \in \{1 \ldots n-2\}$ comme on en a déjà inséré 2 et enlevé 1 il en reste finalement $n-i-2$ à insérer qui auront un coût forcément plus grand ou égal que l'arc de coût le plus faible $\left(\min_{j,i \in \{1 \ldots n-1\}} d(i, j)\right)$. Cette fonction peut paraître coûteuse en temps de calcul mais en réalité la plupart des valeurs sont connues a priori, elle demande donc peu de calculs. Sur notre exemple on a :

$\min_{j \in \{1 \ldots n-1\}} d(j, 0) = 20$ et $\min_{j,i \in \{1 \ldots n-1\}} d(i, j) = 45$

seul $\min_{j \in \{1 \ldots n-1\}} d(v_i, j)$ est à calculer au cours de l'algorithme.

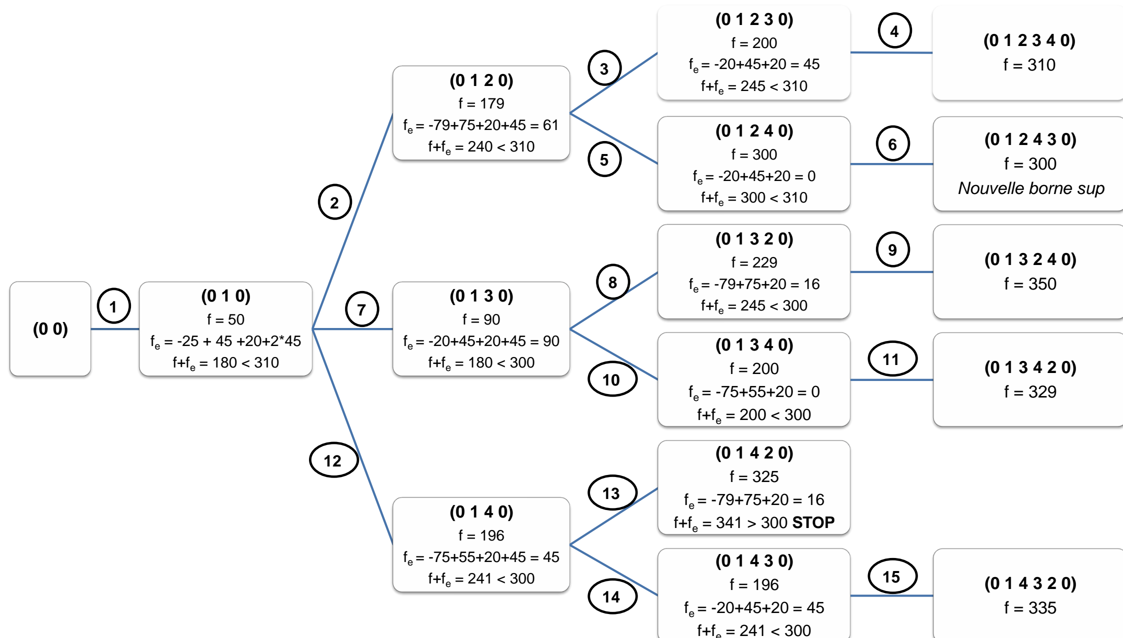


FIG. 1.11 – Illustration d'un *branch and bound* sur la première branche d'un arbre construit pour le problème du voyageur de commerce

Simulons une exécution de l'algorithme avec une exploration en profondeur sur la branche 1 à partir de la solution partielle (0 1 0). L'algorithme complet explore bien sûr toutes les branches mais comme l'exemple est long et répétitif, nous nous contenterons d'une seule branche (figure 1.11). On note dans les nœuds de l'arbre f le coût de la solution partielle et f_e le coût de complétion minimal. On numérote l'ordre d'exploration des nœuds, ce qui permet de voir les descentes jusqu'aux feuilles et les remontées dans les branches. On remarque que pour ce problème l'algorithme n'est pas très efficace puisqu'il ne permet d'éviter la génération que d'une seule solution, ceci est dû au fait que la taille de l'instance traitée est très petite et le problème peu contraint.

1.3.1.2 Elagage de l'arbre de manière heuristique

La méthode du *branch and bound* est très utilisée pour construire des algorithmes exacts. Cependant, il n'est pas toujours possible de trouver un mécanisme de filtrage par évaluation suffisamment efficace et les temps de calcul peuvent être rédhibitoires. Afin de réduire ces temps de calcul une solution est d'utiliser des procédés de filtrage heuristiques, qui sélectionnent seulement une partie des fils de chaque nœud, en utilisant un critère de sélection donné. Ceci limite, généralement, de manière radicale l'explosion combinatoire du nombre de solutions. On peut, par exemple, garder uniquement les m meilleurs fils d'un nœud. La figure 1.12 montre l'arbre qu'on obtiendrait avec $m = 2$ à la place de l'arbre de la figure 1.10. On remarque que les branches partant des nœuds (0 2 0) et (0 4 0) ont été élaguées car leur coût respectif est 158 et 150 alors que les coûts de (0 1 0) et (0 3 0) sont respectivement 50 et 40. Les branches partant de (0 3 2 0) et (0 1 4 0) ont également été élaguées pour les mêmes raisons. On obtient ainsi un arbre beaucoup plus petit, son parcours sera donc bien plus rapide. La solution optimale (0 1 2 4 3 0) a été conservée par ce filtrage heuristique.

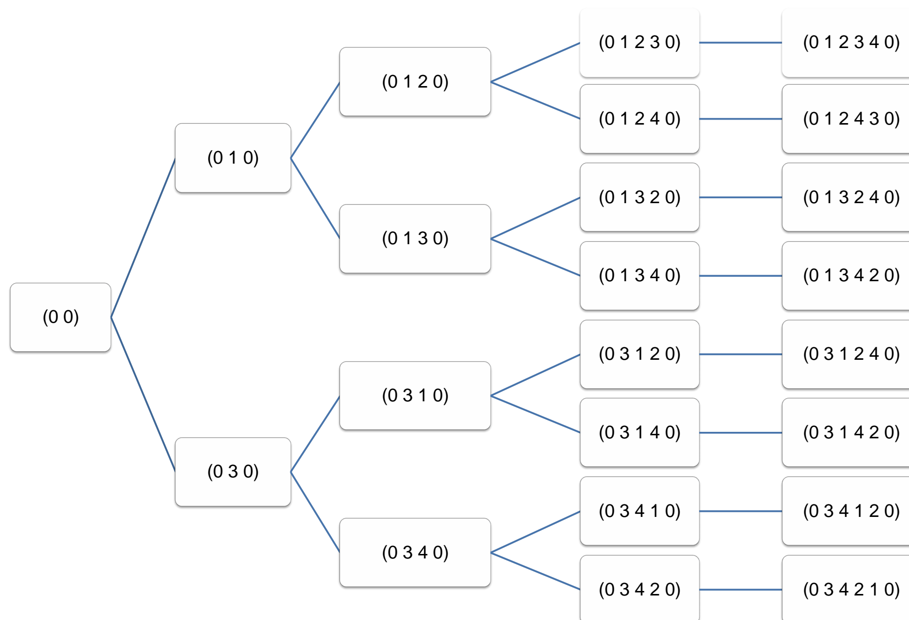


Fig. 1.12 – Arbre d'exploration des solutions du problème de voyageur de commerce dans lequel on ne garde que les 2 meilleurs fils de chaque nœud

1.3.2 La programmation linéaire

La programmation linéaire est un outil classique de la recherche opérationnelle qui peut s'adapter à un grand nombre de problèmes différents. Elle s'applique à la résolution d'un programme linéaire. On appelle programme (ou modèle) linéaire un modèle mathématique représentant un problème d'optimisation, dans lequel on cherche à optimiser une fonction objectif linéaire sous un certain nombre de contraintes. Ces contraintes sont modélisées par des équations ou inéquations linéaires. Il existe des méthodes qui assurent la résolution exacte d'un tel programme. Une des méthodes les plus connues est la méthode du simplex (de son inventeur G.B. Dantzig), en théorie, elle a une complexité non polynomiale. Cependant, il s'avère qu'elle est très efficace en pratique. Comme toutes les techniques de résolution exacte, les temps de calcul peuvent vite devenir très grands, il est courant d'utiliser des techniques supplémentaires comme la génération de colonnes ou le *branch and cut* (coupes et branchements) pour les problèmes de grande taille. Des librairies d'optimisation comme CPLEX (d'IMB ILOG) implémentent ces méthodes.

Le terme programmation linéaire suppose que les solutions à trouver doivent être représentées par des variables réelles. Il existe aussi la programmation linéaire en nombres entiers dans laquelle les variables sont entières et la programmation linéaire mixte dans laquelle les variables peuvent être entières ou réelles. Notons que la résolution d'un programme linéaire en nombres entiers (ou mixtes) est nettement plus difficile que la résolution d'un programme linéaire à variables réelles (à cause des contraintes d'intégrité supplémentaires). D'autre part, il existe généralement plusieurs modélisations possibles par programmation linéaire d'un problème et il est très difficile de déterminer a priori le meilleur modèle.

A titre d'exemple, formulons un programme linéaire en nombres entiers pour le problème du voyageur de commerce. Pour cela on se munit d'un graphe $G = (X, E)$ complet non orienté. L'ensemble X est l'ensemble des sommets et l'ensemble E est l'ensemble des arcs, les arcs sont valués par des coûts (longueur de l'arc). Un tel graphe est représenté figure 1.13 pour le problème du voyageur de commerce à 5 villes utilisé dans ce chapitre (dont l'instance est donnée tableau 1.1).

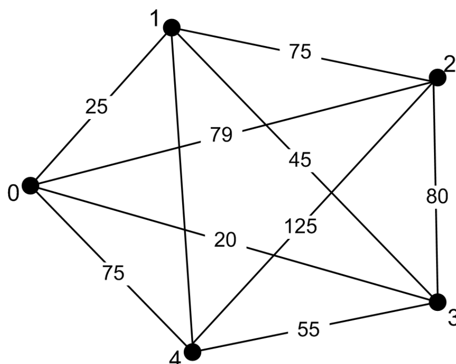


FIG. 1.13 – Graphe associé à l'instance du TSP du tableau 1.1

Il nous faut également introduire quelques notations supplémentaires :

1. pour tout arc (x, y) dans E on note d_{xy} sa longueur ;

2. on associe à chaque arc (x, y) dans E la variable de décision z_{xy} qui prend la valeur 1 si l'arc (x, y) est dans la solution, 0 sinon.

Rappelons que le but du problème du voyageur de commerce est de trouver un cycle hamiltonien (cycle passant une et une seule fois par tous les sommets) de longueur minimale dans un graphe complet valué. Il vient alors le programme linéaire $\mathcal{PL}$ donné figure 1.14.

$$\mathcal{PL}\begin{cases} \textbf{Minimiser} \sum_{(x,y)\in E} d_{xy}.z_{xy} & (1) \\ \forall x \in X, \quad \sum_{y\in X} z_{xy} = 1 & (2) \\ \forall x \in X, \quad \sum_{y\in X} z_{yx} = 1 & (3) \\ \forall Y \subsetneq X, \quad \sum_{x\in Y, y\in Y} z_{xy} < |Y| & (4) \\ \forall (x,y) \in E, \quad z_{xy} \in \{0,1\} & (5) \end{cases}$$

FIG. 1.14 – Programme linéaire pour le TSP

L'objectif et les contraintes du programme linéaire $\mathcal{PL}$ sont les suivants :

- Le coût total de la solution est la somme des longueurs des arcs (x, y) qui la composent, ce qui s'écrit à l'aide des variables de décision $\sum_{(x,y)\in E} d_{xy}.z_{xy}$. L'objectif est de minimiser ce coût d'où l'objectif (1) ;
- Un sommet $x \in X$ est visité une et une seule fois donc il existe un unique arc entrant en x : $\sum_{y\in X} z_{yx} = 1$, de même il existe un unique arc sortant de x : $\sum_{y\in X} z_{xy} = 1$, d'où les contraintes (2) et (3) ;
- Un sous-ensemble de sommets de X ne doit pas définir de cycle donc pour tout sous-ensemble Y de X tel que $Y \neq X$, le nombre d'arcs ayant ses deux extrémités dans Y doit être strictement plus petit que le cardinal de Y (voir l'exemple de la figure 1.15), d'où la contrainte (4) $\sum_{x\in Y, y\in Y} z_{xy} < |Y|$. Remarquons que le nombre de contraintes de ce type est exponentiel ;
- Les variables de décision z ne peuvent prendre que les valeurs 0 ou 1, d'où la contrainte (5).

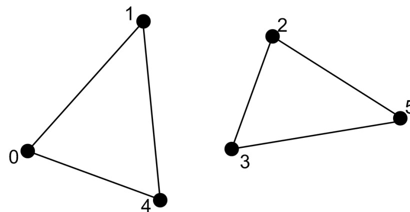


FIG. 1.15 – Les sous-ensembles de sommets $Y_1 = \{0, 1, 4\}$ et $Y_2 = \{2, 3, 5\}$ de l'ensemble $X = \{0, 1, 2, 3, 4, 5\}$ définissent des cycles, on peut vérifier qu'ils violent la contrainte (4) du programme linéaire $\mathcal{PL}$

Après optimisation, une solution du problème est donnée par les variables de décision z_{xy} de valeur non nulle. Par exemple, pour la solution (0 1 2 4 3 0) de la figure 1.3(d) on a $z_{01} = 1$, $z_{12} = 1$, $z_{24} = 1$, $z_{43} = 1$, $z_{30} = 1$, toutes les autres variables de décision étant de valeur nulle. Notons que l'ordre de parcours de ce cycle n'a pas d'importance donc une solution est effectivement entièrement définie par la donnée des arcs qui la composent sans notion d'ordre sur ces arcs.

1.4 Conclusion

Dans ce chapitre, nous nous sommes intéressé à des problèmes d'optimisation combinatoire et nous avons passé en revue différentes approches de résolution. Tout d'abord, nous avons vu qu'il est possible d'établir un classement en fonction de la complexité des algorithmes exacts qui les résolvent. Ensuite, deux grandes approches de résolution ont été envisagées. D'une part, les heuristiques et métaheuristiques fournissent des algorithmes de résolution approchée sans garantir la qualité de la solution. D'autre part, les méthodes exactes telles que l'exploration arborescente ou la programmation linéaire, permettent de calculer des solutions optimales mais avec des temps de calcul pouvant être prohibitifs. Pour être efficaces en temps ces méthodes doivent être couplées à des techniques sophistiquées (méthode de coupe, méthode de branchement, bornes inférieures ...).

Les méthodes exactes sont souvent difficiles à mettre en œuvre sur des problèmes de grande taille et s'adaptent difficilement à de nouvelles contraintes. C'est pourquoi, dans les chapitres suivants, nous nous intéressons à la résolution approchée de différents problèmes *NP-complets*. Le chapitre 2 est dédié à un problème classique d'ordonnancement sous contraintes de ressources : le RCPSP (*Resource-Constrained Project Scheduling Problem*). Il est tout d'abord présenté sous sa forme basique, pour laquelle nous proposons une heuristique basée sur une modélisation multiflot. Nous traitons ensuite une extension originale qui considère des contraintes temporelles additionnelles que l'on nomme *time lags conditionnels*. Elles est résolue grâce à l'heuristique précédente, dans laquelle nous avons intégré la gestion de ces nouvelles contraintes. Nous envisageons également de prendre en compte des contraintes financières qui donnent lieu a des flux financiers entre les activités. Nous évoquons, pour cette extension, les principales modifications à apporter aux algorithmes initiaux. Enfin, nous proposons une approche de résolution originale pour un problème de placement orthogonal en deux dimensions. Nous montrons comment ce problème peut être relaxé en un RCPSP et nous utilisons la solution du RCPSP pour construire la solution du problème de placement. Le chapitre 3 présente deux problèmes de tournées de véhicules avec ramassage et livraison (*Pickup and Delivery Problems*). Le premier, inspiré des mouvements des grues portiques, s'appelle le *Stacker Crane Problem*. Nous proposons, pour la version préemptive de ce problème, une approche de résolution originale à l'aide d'une représentation des tournées sous forme d'arbres. Le second est un problème de transport à la demande (*Dial-a-Ride Problem*) dans lequel nous intégrons des contraintes financières. Nous avons implémenté différentes heuristiques, entre autre, nous verrons dans ce chapitre une implémentation efficace d'une exploration arborescente heuristique. Le chapitre 4 présente un problème qui mélange tournées de véhicules et ordonnancement. Il s'agit du

2L-CVRP dans lequel des colis en deux dimensions doivent être livrés à des clients. La résolution de ce problème met en jeu de nombreuses techniques évoquées dans ce chapitre (heuristique de construction, hybridation de métaheuristiques, méthode de pénalités, ...) et utilise les résultats du chapitre 2 concernant le problème de placement orthogonal.

Chapitre 2

Problèmes d'ordonnancement sous contraintes de ressources

Ce chapitre concerne les problèmes d'ordonnancement et plus particulièrement les problèmes d'ordonnancement de projet sous contraintes de ressources (RCPSP en anglais). Ce type de problèmes ont été très étudiés et constituent des problèmes classiques de recherche opérationnelle. Nous proposons dans ce chapitre des méthodes de résolution pour le RCPSP, pour des extensions du RCPSP et pour un problème de placement qui peut être relaxé en RCPSP.

2.1 Problèmes d'ordonnancement de projet sous contraintes de ressources

Les problèmes d'ordonnancement sous contraintes de ressources sont courants dans le cadre de gestion de projets et font partie des problèmes classiques de recherche opérationnelle. Dans cette section, nous donnons tout d'abord un aperçu des problèmes d'ordonnancement les plus connus. Nous évoquons ensuite les différentes approches qui ont déjà été étudiées pour le RCPSP. Enfin, nous présentons quelques résultats classiques pour le RCPSP : formulation à l'aide de flots, représentations graphiques d'une solution et techniques classiques de résolution.

2.1.1 Aperçu des problèmes d'ordonnancement

2.1.1.1 L'ordonnancement d'atelier

Les problèmes d'ordonnancement d'atelier sont constitués d'un ensemble d'opérations à ordonnancer et d'un ensemble de machines : des pièces doivent subir un ensemble d'opérations sur des machines. On appelle souvent *job* l'ensemble des opérations que doit subir une pièce. Le but est de trouver un ordonnancement des opérations qui minimise le makespan, c'est-à-dire, le temps écoulé entre le début et la fin des opérations pour l'ensemble des jobs.

Les contraintes sont de deux types :

– les contraintes de ressource : une machine ne peut traiter qu'une opération à la fois ;

– les contraintes de gamme opératoire : une gamme définit l'ordre des opérations à réaliser pour chaque job.

Les différents problèmes d'ordonnancement d'atelier se distinguent par leur gamme opératoire. On retient trois problèmes principaux :

flow shop : tous les jobs ont la même gamme opératoire. Chaque pièce passe par toutes les machines dans l'ordre défini par la gamme opératoire ;

job shop : la gamme opératoire varie d'un job à l'autre mais l'ordre des opérations pour chaque job est imposé. Chaque pièce passe par toutes les machines dans l'ordre défini par sa gamme opératoire ;

open shop : la gamme opératoire varie d'un job à l'autre et l'ordre des opérations pour un job n'est pas imposé. Dans ce problème, on doit en plus, déterminer l'ordre des opérations de chaque job.

Il existe également un problème appelé « une machine » dans lequel une unique opération est à réaliser sur une unique machine. Ce problème contient des contraintes additionnelles (typiquement des délais qui dépendent de la séquence) qui le rendent non trivial.

Dans les problèmes de *flow shop*, *job shop* et *open shop*, une opération donnée ne peut être réalisée que par une machine. Il existe des formes étendues de ces problèmes dans lesquelles plusieurs machines peuvent réaliser la même opération donnant alors lieu à un problème d'affectation en plus du problème d'ordonnancement. Ces problèmes se nomment *flowshop* hybride, *jobshop* flexible et *openshop* généralisé. Il existe un problème similaire, nommé « machine parallèle », dans lequel on dispose d'un ensemble de machines identiques pour traiter les jobs, ces derniers n'ayant qu'une seule opération à subir.

2.1.1.2 L'ordonnancement de projet

Les problèmes d'ordonnancement de projet sont constitués d'un ensemble de jobs à ordonnancer et d'un ensemble de ressources. Il peut y avoir différents types de ressources, chaque type de ressources étant disponible en quantité limitée. Le traitement d'un job consomme une certaine quantité d'un ou plusieurs types de ressources. Le problème d'ordonnancement de projet classique est le RCPSP (Resource-Constrained Project Scheduling Problem) dans lequel toutes les ressources sont renouvelables et disponibles en quantité fixée. Le RCPSP est un problème qui a été énormément étudié, c'est pourquoi il existe de nombreuses variantes et extensions de ce problème. Dans [HBss], Hartmann et Briskorn dressent une classification des variantes en 5 classes : les variantes qui portent sur les contraintes liées aux activités (préemptivité, consommation de ressources variable en fonction du temps, multi-modes, ...), les variantes liées aux contraintes temporelles (time lags, contraintes sur les dates de début et fin des activités, ...), les variantes liées aux contraintes de ressources (disponibilité des ressources variable en fonction du temps, ressources non renouvelables, ...), les variantes qui portent sur la fonction objectif (optimiser la robustesse du planning, minimiser les coûts dans le cas de ressources payantes, ...) et enfin, une variante qui consiste à planifier plusieurs projets en même temps.

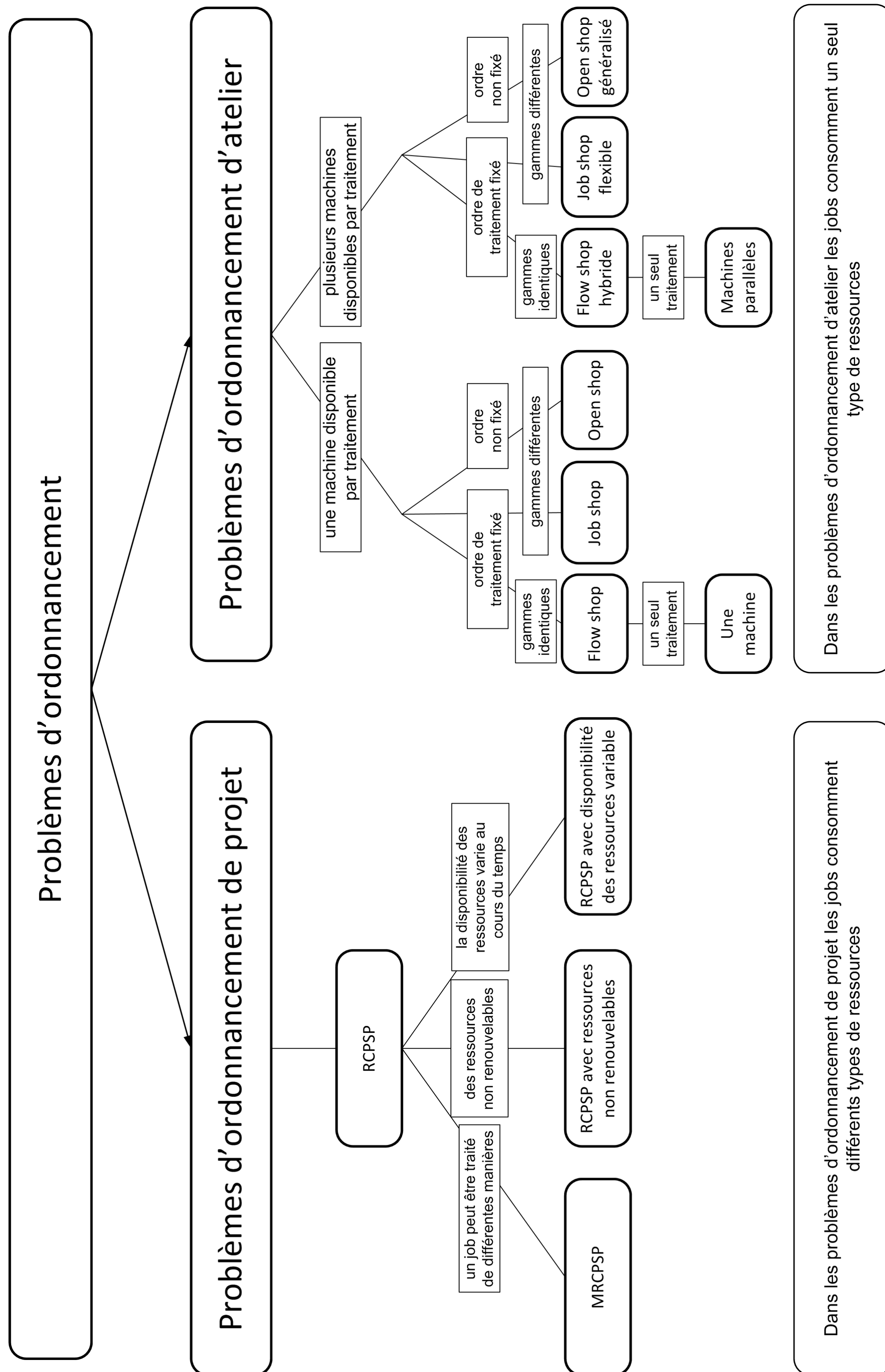


Fig. 2.1 – Problèmes d'ordonnancement les plus connus

Dans le cadre de la comparaison avec les problèmes d'ordonnancement d'atelier, nous présentons les variantes qui concernent les ressources :

RCPSP avec ressource non renouvelable : dans cette extension, il peut exister des ressources non renouvelables (ressources qui ne peuvent être utilisées qu'une seule fois) ;

Multi-mode RCPSP (MRCPSP) : un job peut être traité de différentes manières appelées « modes ». La consommation de ressources et la durée du job sont différents dans chaque mode. Le traitement dans certains modes peut nécessiter la consommation de ressources non renouvelables. La préemption est interdite et une fois le traitement commencé dans un mode donné, il n'est plus possible de changer de mode ;

RCPSP avec ressources à disponibilité variable : la disponibilité des ressources varie en fonction du temps.

La figure 2.1 présente une synthèse des différents problèmes classiques d'ordonnancement .

2.1.2 Aperçu des études et méthodes existantes pour le RCPSP

Il existe de très nombreuses publications concernant le RCPSP, il n'est donc pas possible de fournir une liste exhaustive de tous les travaux déjà réalisés sur ce problème. Nous donnons ici un aperçu général des études et méthodes existantes.

Il existe de nombreuses études et classifications concernant le RCPSP (on peut citer par exemple [KP01], [HDR99], [BDM+99], [Bap95], [BLLN06], [CC88]). Ce problème est associé à diverses applications pratiques dans la planification d'activités industrielles (voir [BESW94], [BLLN06], [Her05], [LW08]). Il existe également de nombreuses études théoriques qui reposent sur l'utilisation d'outils mathématiques comme la programmation linéaire, la programmation quadratique, et des outils comme des structures d'ensembles partiellement ordonnées ou des hypergraphes ([Dam05], [DQS07], [DM41], [FG65]).

De nombreuses extensions au RCPSP ont été étudiées. Elles incluent, par exemple, la préemption des activités ([DQS07], [MQ05], [MQ97], [MC70]), des contraintes temporelles nommées time lags ([RH98]), des ressources non renouvelables ([Kim01]), des flux financiers ([Kim01], [LW08]), la robustesse liée à l'incertitude ([CH08]), ou encore des ressources redondantes ([CN07]). Il existe également des extensions dans lesquelles le critère de minimisation n'est pas la durée du projet. La minimisation peut être liée, par exemple, à des coûts économiques ou à des pénalités de retard (voir [HG03], [ASA06]).

Le RCPSP est un problème NP-complet quelque soit la variante considérée, plusieurs formulations ont été données (voir [Bap95], [CC88], [LLRKS93]). En pratique, l'obtention de résultats exacts devient difficile à partir de 60 activités et 4 ressources environ ([KH99], [DH97]). La génération d'instances de tests pour ce problème et la caractérisation de leur complexité peut être également vu comme un problème difficile ([KSD95], [KH99]). De nombreuses approches de résolution exacte du RCPSP utilisent la programmation linéaire ([Dam05], [Kim01], [MMRB98]), ou encore des techniques d'exploration arborescente avec séparation-évaluation (*branch & bound*), de génération de coupes ou de propagation de

contraintes ([Dam05], [RH98], [Pat84], [BKST98]). Il existe également des méthodes basées sur des extensions d'algorithmes pour le problème d'ordonnancement multi-machines (*multiprocessor scheduling*) : voir [MQ05], [Dje99], [JMRW04]. Des bornes inférieures de bonne qualité peuvent être obtenues par l'application de génération de colonnes sur des programmes linéaires spécifiques, par des processus de raisonnements énergétiques, par des considérations sur la préemption des activités ou encore par le calcul de plus longs chemins (voir [DQS07], [BD04], [BK00], [CN03]). Des heuristiques très efficaces ont également été développées : algorithmes gloutons basés sur des règles de priorité ([CH08], [MC70]), algorithmes de flots et techniques d'insertion ([AR00], [AMR03]). De nombreuses métaheuristiques ont été utilisées pour ce problème comme la recherche tabou ([KS03], [NI02], [TL00], [BBK98]), les algorithmes génétiques ([VBQ03], [VBQ05], [AMR04], [Har02], [CT03]), la *scatter search* ([DRLV06]), le recuit simulé ([BL03a]).

2.1.3 Définition et modélisation du RCPSP

2.1.3.1 Définition du problème et notations

Un RCPSP est constitué d'un ensemble d'activités $A = \{1, \ldots, n\}$ à planifier, et d'un ensemble de ressources $R = \{1, \ldots, m\}$. Chaque type de ressources $k \in R$ est renouvelable (une activité utilise des ressources pendant la durée de son traitement et les restitue ensuite) et disponible en quantité M_k. Chaque activité $i \in A$ consomme r_{ik} unités de ressource de type k et dure d_i unités de temps. Certaines activités i, j, peuvent être liées par des contraintes de précédence, notées $i \ll j$, signifiant que j doit commencer après la fin de i. Le problème de planification consiste alors à déterminer la date de début Δ_i de chaque activité i, de sorte que les contraintes de ressources et les contraintes de précédence soient respectées et que la durée totale du projet (makespan), notée C_{max}, soit minimale.

2.1.3.2 Modèle linéaire

Le programme linéaire présenté ici repose sur la modélisation des échanges de ressources entre activités. Ces échanges sont habituellement représentés par des flots. On note f_{ijk} le nombre de ressources de type k transférées directement de l'activité i à l'activité j (l'activité i libère les ressources une fois qu'elle est terminée et toutes les unités d'une même ressource sont équivalentes, connaître le nombre d'unités de ressource transférées est donc suffisant). La valeur de f_{ijk} est donc égale à la quantité du flot k qui transite de i vers j. On note E l'ensemble des arcs (i, j) qui représentent une contrainte de précédence $i \ll j$, pour i et j dans A. Nous ajoutons deux activités fictives s et p telles que $\forall k \in R, r_{sk} = r_{pk} = M_k$ et $d_s = d_t = 0$. On introduit une variable binaire x_{ij} qui vaut 1 si i précède j (c'est le cas si $i \ll j$, ou s'il y a un transfert de ressources de i vers j), 0 sinon. Enfin, on considère une constante suffisamment grande que l'on note N (par exemple $N = \sum_{i \in A} d_i + \sum_{k \in R} M_k$).

On peut alors formuler le programme linéaire $\mathcal{P}$, donné figure 2.2 ([AMR03]), qui fait apparaître les contraintes suivantes :

(1) objectif ;

(2) contraintes de précédence ;

(3) dans le cas où $x_{ij} = 1$ ($i \ll j$ ou quantité de flot non nul de i vers j) cette contrainte se réécrit $\Delta_j - \Delta_i \geq d_i$ ce qui signifie que j doit commencer après la fin de i,
dans le cas où $x_{ij} = 0$ cette contrainte se réécrit $\Delta_j - \Delta_i \geq d_i - N$, la valeur de N implique qu'elle est trivialement vérifiée ;

(4) dans le cas où $x_{ij} = 1$ cette contrainte se réécrit $f_{ijk} - N \leq 0$, la valeur de N implique qu'elle est trivialement vérifiée,
dans le cas où $x_{ij} = 0$ cette contrainte se réécrit $f_{ijk} \leq 0$ ce qui impose une quantité de flot nulle entre i et j ;

(5) cette contrainte assure que chaque activité i fournit, pour chaque ressource k, une quantité de flot sortant égale à sa consommation de ressource k ;

(6) cette contrainte assure que chaque activité j reçoit pour chaque ressource k une quantité de flot entrant égale à sa consommation de ressource k ;

(7) correspond à l'expression classique du makespan : C_{max} donne la date à laquelle toutes les activités sont terminées ;

(8) les quantités de flot sont des entiers positifs ou nuls ;

(9) les variables x_{ij} sont binaires : elles ne peuvent prendre que la valeur 0 ou la valeur 1.

$$\mathcal{P} : \begin{cases} \textbf{Minimiser } C_{max} & & (1) \\ \forall (i,j) \in E, & x_{ij} = 1 & (2) \\ \forall i \in A \cup \{s\},\ \forall j \in A \cup \{p\}, & \Delta_j - \Delta_i - N x_{ij} \geq d_i - N & (3) \\ \forall i \in A \cup \{s\},\ \forall j \in A \cup \{p\},\ \forall k \in R, & f_{ijk} - N x_{ij} \leq 0 & (4) \\ \forall i \in A \cup \{s\},\ \forall k \in R, & \sum_{j \in A \cup \{p\}} f_{ijk} = r_{ik} & (5) \\ \forall j \in A \cup \{p\},\ \forall k \in R, & \sum_{i \in A \cup \{s\}} f_{ijk} = r_{jk} & (6) \\ \forall i \in A, & C_{max} \geq \Delta_i + d_i & (7) \\ \forall i \in A \cup \{s\},\ \forall j \in A \cup \{p\}, \forall k \in R, & f_{ijk} \in \mathbb{N} & (8) \\ \forall i \in A \cup \{s\},\ \forall j \in A \cup \{p\}, & x_{ij} \in \{0; 1\} & (9) \end{cases}$$

FIG. 2.2 – Programme linéaire pour le RCPSP

2.1.4 Les techniques classiques de représentation

2.1.4.1 Représentation d'une solution à l'aide d'un graphe

Comme pour tous les problèmes d'ordonnancement, une solution d'un RCPSP peut être efficacement représentée par un graphe. Les nœuds sont les activités de $A \cup \{s, p\}$. Il existe un arc orienté de l'activité i vers l'activité j s'il existe une ressource $k \in R$ tel que $f_{ijk} > 0$ ou si $i \ll j$. Chaque arc (i, j) est valué par le poids d_i (durée de l'activité i), et associé à

un vecteur de flots $F_{ij} = (f_{ij1}, \ldots, f_{ijm})$ qui représente le transfert de ressources entre i et j.

Ce graphe peut être utilisé pour calculer les dates de début au plus tôt (*earliest starting time* ES_i) et au plus tard (*latest starting time* LS_i) auxquelles peut commencer une activité i. La date de début au plus tôt correspond à la date avant laquelle l'activité ne peut commencer compte tenu des contraintes de flots et de précédence. La date de début au plus tard correspond à la date après laquelle l'activité ne peut commencer sans que cela n'ait un impact sur la durée totale du projet. Dans le graphe, la date de début au plus tôt est donnée par la longueur $l(s, i)$ du plus long chemin entre s et i et la date au plus tard est donnée par $l(s, p) - l(i, p)$. On appelle chemin critique, le chemin dans le graphe constitué des activités dont la date de début au plus tard est égale à la date de début au plus tôt. Le makespan est la longueur $l(s, p)$ du chemin entre la source et le puits.

Notons qu'un tel graphe ne doit pas contenir de circuit, auquel cas il n'y a pas existence de plus long chemin.

Exemple 4. *Considérons un problème à 6 activités et 2 ressources (ressource 1 disponible en quantité 5 et ressource 2 disponible en quantité 4) dont l'instance est donnée tableau 2.1. Une solution de ce problème est proposée tableau 2.2 et le graphe associé est donné figure 2.3.*

activité	durée	consommation ressource 1	consommation ressource 2	prédécesseurs
1	2	5	0	-
2	3	3	1	-
3	2	1	1	-
4	2	4	1	2,3
5	1	0	3	2
6	3	3	2	4

TAB. 2.1 – Exemple d'instance pour un RCPSP à 6 activités, 2 ressources

activité	début au plus tôt	début au plus tard
1	0	0
2	2	2
3	2	3
4	5	5
5	5	6
6	7	7

TAB. 2.2 – Exemple de solution (date de début des activités) pour l'instance du tableau 2.1

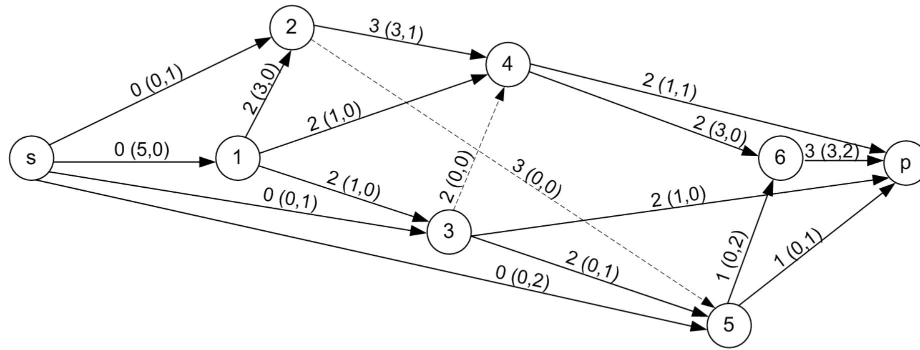


FIG. 2.3 – Représentation par un graphe de la solution du tableau 2.2, les flèches en pointillés représentent une contrainte de précédence, un arc (i, j) est labellisé par la durée de i suivie du multiflot F_{ij}

2.1.4.2 Représentation d'une solution à l'aide d'un diagramme de Gantt

Le graphe permet de visualiser les échanges de ressources entre les activités et les contraintes de précédence, cependant il ne permet pas de visualiser facilement les dates de début des activités. Un outil très utilisé pour visualiser la planification des activités est le diagramme de Gantt, il peut prendre différentes formes suivant le problème d'ordonnancement traité. Dans le cadre du RCPSP, le diagramme de Gantt consiste à placer les activités, représentées par des rectangles, dans un repère en deux dimensions dont l'axe des abscisses représente le temps et l'axe des ordonnées représente la quantité de ressources consommées. On dresse un diagramme par type de ressources. Pour une ressource $k \in K$ les activités sont schématisées par des rectangles de longueur leur durée et de hauteur leur consommation en ressource k. On place alors ces rectangles dans le repère de sorte que l'abscisse du coin inférieur gauche soit égale à la date de début de l'activité. Une solution du RCPSP est entièrement définie par les dates de début des activités, donc seul l'abscisse des rectangles est imposée, mais il est commode de fixer les ordonnées de manière à représenter les échanges de ressources : le nombre d'unités en ordonnée, communes à deux rectangles consécutifs i et j, est alors égale à la quantité de ressources k transmises de l'activité i vers l'activité j. Il est clair que pour une solution, il existe plusieurs manières de positionner les rectangles qui respectent ces deux conditions. Une solution n'est donc pas associée à une représentation unique.

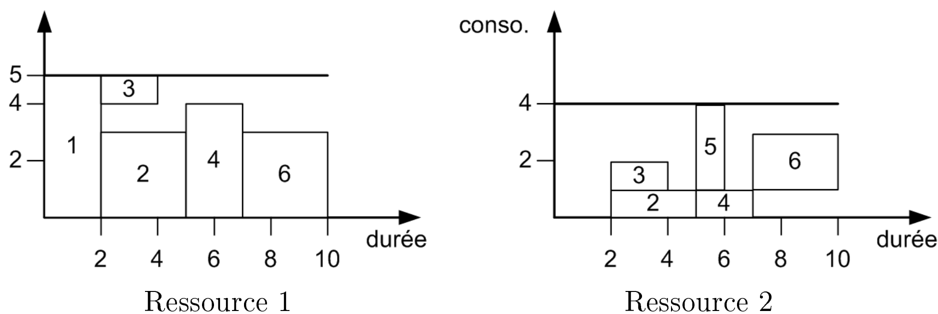


FIG. 2.4 – Exemple de diagrammes de Gantt (au plus tôt) pour chacune des ressources

La figure 2.4 montre un exemple de diagrammes de Gantt correspondant à la solution du tableau 2.2 où les activités sont planifiées avec leur date au plus tôt, ces diagrammes respectent les flots du graphes 2.3. De même, la figure 2.5 illustre cette solution avec les dates au plus tard.

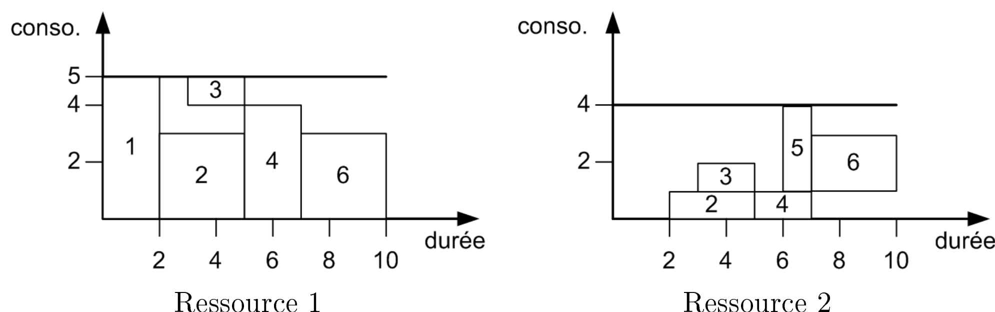


FIG. 2.5 – Exemple de diagrammes de Gantt (au plus tard) pour chacune des ressources

2.1.5 Les techniques classiques de résolution

Le RCPSP étant un problème ancien et très étudié, plusieurs algorithmes sont devenus classiques. Nous présentons ici un schéma heuristique général de résolution et un voisinage qui s'appuie sur les échanges de ressources.

2.1.5.1 Heuristique de planification basée sur des règles de priorité

Une heuristique de planification basée sur des règles de priorité possède deux composants : un schéma de génération d'ordonnancements (*Schedule Generation Schemes* - SGS) et une règle de priorité. Nous présentons ici l'heuristique de manière générale car il existe de nombreuses règles de priorité (voir [Kol96a]).

Le schéma de génération d'ordonnancements est une méthode gloutonne qui travaille sur des ordonnancements partiels : initialement aucune activité n'est ordonnancée, à chaque étape une nouvelle activité est insérée dans l'ordonnancement partiel. L'idée est de commencer l'ordonnancement à la date $t = 0$ et de faire évoluer cette date au cours des itérations. Ainsi à une itération k, on associe la date t_k, l'ensemble Ac_k des activités actives en t_k (qui débutent avant t_k et terminent après) et l'ensemble D_k des activités qui peuvent commencer en t_k compte tenu des ressources encore disponibles et des contraintes de précédence. La décision prise à l'itération k n'est jamais remise en cause : une activité insérée à l'itération k voit sa date de début (au plus tôt) fixée à t_k. Le fonctionnement général de cette heuristique est décrit par l'algorithme 12. On rappelle que pour une activité j la variable ES_j désigne sa date de début au plus tôt et d_j désigne sa durée. Notons qu'on peut sans difficulté, au cours de cet algorithme, tenir à jour les quantités de ressources transférées entre les activités de manière à pouvoir construire le graphe de la solution avec les échanges de ressources.

Algorithme 12 : schéma de génération d'ordonnancements (SGS)

Entrées : $A = \{1, \ldots, n\}$ // ensemble d'activités

1 $t_k = 0$;
2 $Ac_k = \emptyset$;
3 $D_k = \{i \in A, \nexists j \in A, j \ll i\}$;
4 $k = 1$;
5 **Tant que** $k \leq n$ **faire**
6   **Calculer** la priorité des activités de D_k ;
7   **Choisir** dans D_k l'activité j qui a la meilleure priorité ;
8     $ES_j = t_k$ //j commence en t_k ;
9   **Mettre à jour** les ensembles Ac_k et D_k ;
10   **Tant que** $D_k = \emptyset$ **faire**
11         $t_k \leftarrow \min_{j \in Ac_k} (ES_j + d_j)$;
12   **Mettre à jour** les ensembles Ac_k et D_k ;
13     $k = k + 1$;

2.1.5.2 Le voisinage : modification « parallèle » du flot

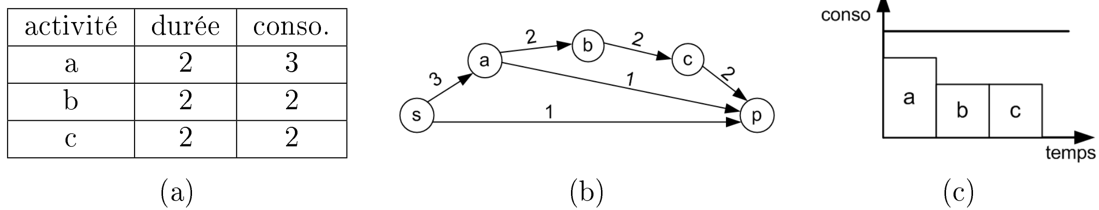
Nous présentons un voisinage classique basé sur la modification de flot (voir [FH97]), le but étant simplement de donner au lecteur une idée des techniques couramment utilisées dans le cadre de la résolution de RCPSP.

Dans un voisinage de type parallèle, l'idée est de réorganiser les échanges de ressources entre activités afin de permettre à une ou plusieurs activités de commencer plus tôt. Pour cela, on tente de supprimer un arc du chemin critique : on choisit un arc (i, j) du chemin critique (qui existe à cause d'un échange de ressources et non d'une contrainte de précédence) et on essaie de dévier la quantité de flot présente sur cet arc. On utilise pour cela les arcs qui existent entre les prédécesseurs de i et les successeurs de j (voir algo 13). On rappelle que F_{ij}=$(f_{ij1}, \ldots, f_{ijm})$ désigne la quantité de flot de i vers j.

Algorithme 13 : modification « parallèle » du flot

1 **Choisir** un arc (i, j) du chemin critique ;
2 $stop \leftarrow faux$;
3 **Tant que** $F_{i,j} \neq 0$ *et* $stop = faux$ **faire**
4   **Choisir** un arc $(pred, succ)$ où $pred$ est un prédécesseur de i et $succ$ un successeur de j;
5   **Pour** $k = 1 \ldots m$ *//pour chaque ressource* **faire**
6         $q = \min(f_{ijk}, f_{pred,succ,k})$;
7         $f_{pred,succ,k} = f_{pred,succ,k} - q$;
8         $f_{ijk} = f_{ijk} - q$;
9         $f_{pred,j,k} = f_{pred,j,k} + q$;
10         $f_{i,succ,k} = f_{i,succ,k} + q$;
11   **Si** *il n'existe plus d'arc* $(pred, succ)$ *portant du flot* **alors**
12         $stop \leftarrow vrai$;

Considérons l'instance à 3 activités et un type de ressource disponible en quantité 4,



activité	durée	conso.
a	2	3
b	2	2
c	2	2

FIG. 2.6 – (a) instance à 3 activités, un type de ressource disponible en quantité 4 ; (b) et (c) exemple de solution (flot et Gantt)

donnée figure 2.6(a) et la solution proposée figure 2.6(b) associée au diagramme de Gantt figure 2.6(c). Pour plus de lisibilité, on ne fait apparaître que le flot sur les arcs. Pour améliorer la solution, il faut supprimer un arc du chemin critique par exemple (b, c). Pour cela, on considère les prédécesseurs de b (ici a et s) et les successeurs de c (ici p). On applique l'algorithme 13 avec $(pred, succ) = (a, p)$, puis avec $(pred, succ) = (s, p)$ (comme il n'y a qu'une ressource on note simplement f_{ij} le flot de i vers j) : à l'itération 1 on a

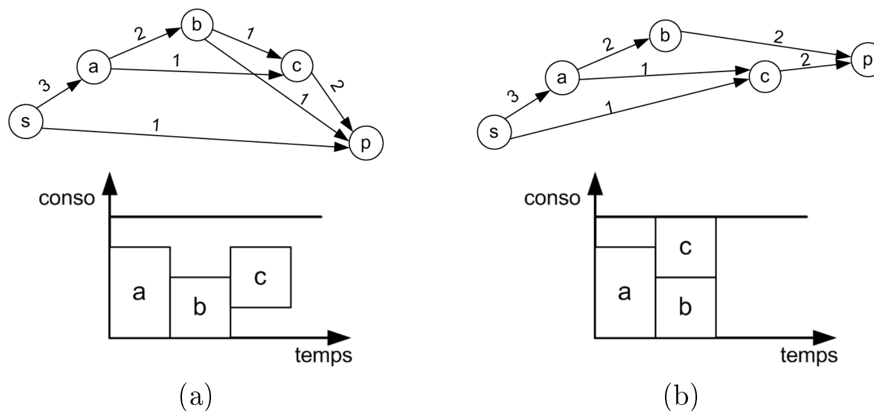
$q \leftarrow min(f_{bc}, f_{ap}) = 1$;
$f_{ap} \leftarrow f_{ap} - 1 = 0$;
$f_{bc} \leftarrow f_{bc} - 1 = 1$;
$f_{ac} \leftarrow f_{ac} + 1 = 1$;
$f_{bp} \leftarrow f_{bp} + 1 = 1$.

A l'itération 2 on a

$q \leftarrow min(f_{bc}, f_{sp}) = 1$;
$f_{sp} \leftarrow f_{sp} - 1 = 0$;
$f_{bc} \leftarrow f_{bc} - 1 = 0$;
$f_{sc} \leftarrow f_{sc} + 1 = 1$;
$f_{bp} \leftarrow f_{bp} + 1 = 2$.

On a réussi à vider l'arc (bc) $(f_{bc} = 0)$.

Au terme de l'itération 1, on obtient le graphe et le Gantt figure 2.7(a). Au terme de l'itération 2, on obtient le graphe et le Gantt figure 2.7(b).



(a) (b)

FIG. 2.7 – (a) Flot et Gantt après itération 1 ; (b) Flot et Gantt après itération 2

Notons qu'il est impossible de créer des cycles dans le graphe avec ce voisinage car l'ordre des activités reste inchangé, seule l'utilisation des ressources change. Il existe une

modification « série » du flot qui consiste à retourner un arc du chemin critique (voir [FH97]) et, dans ce cas, la séquence des activités est changée.

2.2 Proposition de résolution du RCPSP à l'aide des flots

Dans cette section on s'intéresse à la résolution du RCPSP. Le problème est tout d'abord modélisé à l'aide d'un multiflot, puis résolu de manière heuristique. L'intérêt de cette approche réside dans le fait que notre modèle de flots est facilement généralisable à des extensions du RCPSP comme les *time lags* conditionnels (voir section 2.3). Notre but est donc de concevoir des heuristiques à base de flots pour le RCPSP et de les adapter ensuite à la résolution d'extensions de ce problème.

La théorie des flots ([AMO93], [MMRB98], [AMOR95]) est très utilisée dans la modélisation et la conception d'algorithmes qui mettent en jeu la circulation de biens, de personnes, d'argent, d'énergie ou d'information. Elle a été essentiellement utilisée pour optimiser des systèmes de transport et de télécommunication ([DS94]) ou des réseaux de distribution de gaz et d'électricité ([AMOR95], [PD98], [MF90]). Elle s'avère être un outil très puissant non seulement pour la modélisation de ces problèmes, mais aussi pour faire le lien entre la programmation linéaire et les techniques purement combinatoires. De ce fait, de nombreuses recherches portent sur les problèmes de flots et la prise en compte de contraintes combinatoires ([PD98], [CW81], [Ame00], [BMM98]). Nous proposons ici une application des flots à un problème d'ordonnancement, les flots permettant de représenter facilement les échanges de ressources.

2.2.1 Multiflot associé au RCPSP

2.2.1.1 Rappel des notations

On utilise les notations de la section 2.1.3.1. Une instance du RCPSP est définie par :

- un ensemble d'activités $A = \{1, \dots, n\}$: chaque activité i a une durée d_i ;
- un ensemble de ressources renouvelables $R = \{1, \dots, m\}$: chaque ressource k est disponible en quantité M_k. Une activité $i \in A$ consomme r_{ik} unités de ressource de type k pendant la durée de son exécution ;
- des contraintes de précédence entre les activités : $i \ll j$, signifie que l'activité j doit commencer après la fin de l'activité i.

Le problème consiste alors à calculer la date de début Δ_i de chaque activité $i \in A$ tel que :

- les contraintes de précédence sont respectées :

$$\forall i \in A,\ \forall j \in A,\ i \ll j \Rightarrow \Delta_i + d_i \leq \Delta_j;$$

- les contraintes de ressource sont respectées :

$$\forall k \in R,\ \forall t \in [0, C_{max}],\ \sum_{\substack{i \in A,\\ \Delta_i \leq t < \Delta_i + d_i}} r_{ik} \leq M_k;$$

– le makespan ($C_{max} = \max_{i \in A} (\Delta_i + d_i)$) est minimal.

2.2.1.2 Reformulation du RCPSP en un problème de multiflot

Le but de cette section est de montrer comment une solution du RCPSP peut être représentée par un multiflot. On reprend ici les idées introduites en 2.1.4.1 dans un cadre plus formel qui permet de définir des algorithmes de résolution basés sur les flots.

Soit un graphe $G = (X, E)$, on appelle flot un vecteur f indexé sur les arcs de E qui respecte les lois de Kirchhoff :

$$\forall x \in X, \sum_{y \in X/(x,y) \in E} f_{(x,y)} = \sum_{y \in X/(y,x) \in E} f_{(y,x)}$$

On appelle multiflot un vecteur de flots.

On ajoute deux activités fictives source (s) et puits (p) à l'ensemble d'activités A. On note $A^* = A \cup \{s, p\}$. Les activités s et p sont telles que $\forall k \in R, r_{sk} = r_{pk} = M_k$ et $d_s = d_t = 0$. De plus pour chaque activité $i \in A$ on a $s \ll i$ et $i \ll p$.

Etant donnée une solution du RCPSP, on considère le multiflot $F = (f_1 \dots f_m)$ tel que le flot f_k de F représente le transfert de la ressource k entre les activités de A. On dit alors que F vérifie la propriété P_{flot} définie par :

$$(P_{flot}) \begin{cases} \forall i \in A \cup \{s\},\ \forall k \in R, \sum_{j \in A \cup \{p\}} f_{k(i,j)} = r_{ik} \\ \forall j \in A \cup \{p\},\ \forall k \in R, \sum_{i \in A \cup \{s\}} f_{k(i,j)} = r_{jk} \end{cases}$$

On note $E_{\ll} = \left\{(i,j)/i \in A^*, j \in A^*, i \ll j\right\}$ l'ensemble des arcs induits par une contrainte de précédence et on note $E_F = \{(i,j), i \in A^*, j \in A^*, F_{(i,j)} > 0\}$ l'ensemble des arcs (i, j) qui portent une quantité non nulle de flot. On peut alors représenter la solution d'un RCPSP par un graphe orienté $G = (X, E)$ sans circuit. L'ensemble des sommets X est égal à l'ensemble des activités A^*. Il existe un arc orienté de $i \in X$ vers $j \in X$ s'il y a une contrainte de précédence ou un échange de ressources entre i et j, avec les notations précédentes il vient $E = E_{\ll} \cup E_F$.

Les arcs (i, j) de E sont valués par la durée de i (d_i). Une solution du RCPSP est donc entièrement définie par le graphe G. Les dates de début des activités s'obtiennent par un calcul des plus longs chemins : la date de début au plus tôt d'une activité i est le plus long chemin entre s et i (voir section 2.1.4.1).

2.2.2 Algorithme d'insertion : génération d'une solution initiale

L'algorithme que nous proposons est basé sur des techniques d'insertion originales, cependant, des idées similaires ont été développées par Artigues et *al.* dans [AMR03]. Notre approche est différente car, à chaque nouvelle insertion, les flots dans la coupe considérée sont entièrement recalculés de sorte qu'ils soient optimaux pour cette coupe. De plus, l'objectif de notre approche est d'être facilement réutilisable pour traiter des extensions

du problème initial, elle définie donc des procédés d'insertion plus généraux.

2.2.2.1 Schéma général

Etant donné une instance du RCPSP définie par l'ensemble des activités A, l'ensemble des ressources R et l'ensemble des contraintes de précédence $\ll$, on cherche à calculer un multiflot F qui vérifie P_{flot}. Le multiflot F permet de calculer les arcs E_F, donc de définir le graphe G, donnant ainsi une solution pour l'instance du RCPSP. Le calcul du multiflot F est assuré par un algorithme glouton mettant en jeu un graphe partiel G'. Initialement l'ensemble des sommets de G' est réduit à $\{s, p\}$ et le multiflot F est tel que $F_{(s,p)} = (M_1, M_2, \ldots, M_m)$ (la source s transfère toutes les ressources au puits p). A chaque itération, on ajoute une nouvelle activité dans G' et on met à jour les flots pour que P_{flot} soit toujours respecté.

Exemple 5. *On considère une instance à deux types de ressource R_1 et R_2 et 3 activités, on suppose qu'il n'y a pas de contrainte de précédence. R_1 est disponible en quantité 4 et R_2 est disponible en quantité 6. L'activité 1 consomme 3 R_1 et 2 R_2, l'activité 2 consomme 1 R_1 et 0 R_2 et l'activité 3 consomme 2 R_1 et 5 R_2. On ignore les durées des activités, inutiles pour cet exemple. Le déroulement général de l'algorithme glouton et la mise à jour des flots pour cette instance sont illustrés figure 2.8.*

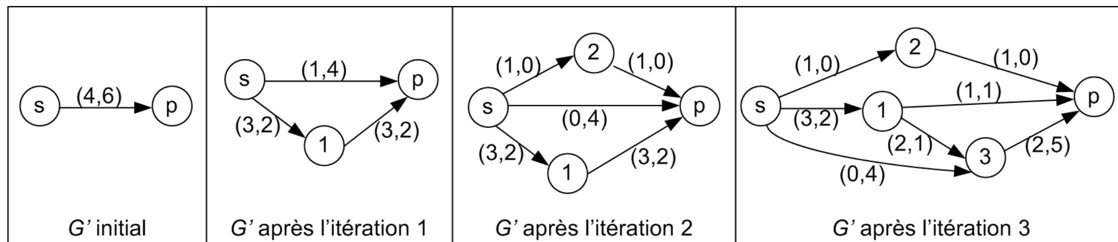


FIG. 2.8 – G' aux différentes itérations de l'algorithme, les arcs portent le multiflot associé

Algorithme 14 : Evaluer (schéma de principe)

Entrées : $A = \{1, \ldots, n\}$, λ //séquence des activités dans un ordre aléatoire
Sorties : F

1 $X \leftarrow \{s, p\}$ // initialisation des sommets de G' ;
2 $\forall k \in R,\ f_{k(s,p)} \leftarrow M_k$ // initialisation du multiflot ;
3 **Tant que** *il reste des activités non insérées* **faire**
4   **Choisir** la prochaine activité x_0 dans λ ;
5   **Calculer** les coupes $(U,\ V)$ compatibles avec l'insertion de x_0 ;
6   **pour chaque** *coupe* $(U,\ V)$ **faire**
7   Calculer le coût d'insertion de x_0 dans (U, V) ;
8   Effectuer l'insertion de plus faible coût et mettre à jour le multiflot F en conséquence ;

Le schéma général de résolution pour le RCPSP est donné par algorithme 14. Les différentes étapes d'insertion d'une nouvelle activité x_0 dans le graphe partiel sont détaillées

dans les sections suivantes. Elles mettent en jeu une coupe (U, V), c'est-à-dire une partition de l'ensemble des sommets en deux sous-ensembles U et V telle qu'il n'y a pas de flot de V vers U. Dans une telle coupe (U, V) on a $s \in U$ et $p \in V$. Ainsi x_0 va recevoir du flot de U et le restituer à V. L'insertion de x_0 se fait en deux étapes, tout d'abord x_0 est « raccroché » à U, ensuite les flots sont calculés au mieux dans la coupe $(U \cup \{x_0\},\ V)$ (*problème de la coupe*). Pour présenter les notations et les problématiques liées à l'insertion d'une activité dans une coupe, on commence par présenter le problème de la coupe, puis le problème de l'insertion dans le cas d'un flot unique. La section 2.2.2.4 étend ces résultats au multiflot.

2.2.2.2 Problème de la coupe

On considère dans cette section un unique flot (donc un seul type de ressources). Dans le cas d'un multiflot (plusieurs types de ressources), il suffit de considérer chaque ressource indépendamment (voir section 2.2.2.4).

Notations

Soit une activité $i \in A^*$ et une coupe (U, V), on note :

- $pred(i)$ les activités qui précédent i à cause d'une contrainte de précédence ou d'un transfert de ressources :

$$pred(i) = \{j \in A^* / j \ll i\} \cup \{j \in A^* / F_{(j,i)} \neq 0\}$$

- $succ(i)$ les activités qui suivent i à cause d'une contrainte de précédence ou d'un transfert de ressources :

$$succ(i) = \{j \in A^* / i \ll j\} \cup \{j \in A^* / F_{(i,j)} \neq 0\}$$

- $\pi(i)$ la date de fin au plus tôt de i, $\pi(i)$ se définit récursivement par :

$$\begin{cases} \pi(s) = 0 \\ \forall i \in A^* - \{s\},\ \pi(i) = \max\limits_{j \in pred(i)} (\pi(j) + d_i) \end{cases}$$

- $\bar{\pi}(i)$ la longueur du plus long chemin entre i et p, $\bar{\pi}(i)$ se définit récursivement par :

$$\begin{cases} \bar{\pi}(p) = 0 \\ \forall i \in A^* - \{p\},\ \bar{\pi}(i) = \max\limits_{j \in succ(i)} (\bar{\pi}(j) + d_i) \end{cases}$$

- $out(i)$ la quantité de flot donnée à la coupe, $out(i)$ est trivialement nulle pour les activités i dans V ;
- $in(i)$ la quantité de flot reçue de la coupe, $in(i)$ est trivialement nulle pour les activités i dans U.

Remarque 1

Les quantités $out(i)$ et $in(i)$ dépendent bien sûr des ensembles U et V, on devrait donc les noter « $out(i, U, V)$ » et « $in(i, U, V)$ ». Cependant, afin de ne pas trop alourdir les notations, nous avons choisi de ne pas faire apparaître les ensembles U et V.

Définition du problème

Soit un graphe biparti (X, E) tel que l'ensemble des sommets X s'écrit $X = U \cup V$. Les sommets de U sont ordonnés dans le sens des π croissants et les sommets de V sont ordonnés dans le sens des $\bar{\pi}$ décroissants. On connaît pour chaque u de U la quantité $out(u)$ et pour chaque v de V la quantité $in(v)$ supposées telles que $\sum_{u \in U} out(u) = \sum_{v \in V} in(v)$. On dit que le flot $h = \big(h_{(u,v)}, u \in U, v \in V\big)$ est compatible avec la coupe (U, V) si :

$$\begin{cases} \forall u \in U,\ out(u) = \sum_{v \in V} h_{(u,v)} \\ \forall v \in V,\ in(v) = \sum_{u \in U} h_{(u,v)} \end{cases}$$

On pose $H = \{u \in U, v \in V / u \ll v \text{ ou } h_{(u,v)} \neq 0\}$.

Pour un flot h compatible avec (U, V) on pose $C'_{max}(h) = \max_{(u,v) \in H} (\pi(u) + \bar{\pi}(v))$. On peut alors définir le problème de la coupe de la manière suivante :

Problème de la coupe : *Etant donné un graphe biparti (X, E) tel que $X = U \cup V$, trouver un flot compatible avec (U, V) qui minimise $C'_{max}(h)$.*

Résolution du problème

Soit une coupe (U, V) et un flot h compatible avec (U, V), on définit pour h la propriété P_{cross} suivante :

$$(P_{cross}) \begin{cases} \text{Il n'existe pas de sommets } u,\ u' \text{ dans } U \text{ et } v, v' \text{ dans } V \text{ tels que :} \\ \quad h_{(u,v)} > 0 \\ \quad h_{(u',v')} > 0 \\ \quad \pi(u') > \pi(u) \text{ et } \bar{\pi}(v') > \bar{\pi}(v) \end{cases}$$

Théorème 1

Etant donné un graphe biparti (X, E) tel que $X = U \cup V$ et un flot h compatible avec (U, V), si h satisfait la propriété P_{cross} alors h est une solution optimale au problème de la coupe.

Démonstration

Soit un flot g compatible avec (U, V) qui satisfait la propriété P_{cross}. On suppose qu'il existe un autre flot h compatible avec (U, V) tel que $C'_{max}(h) < C'_{max}(g)$.

Donc $C'_{max}(g) > \max_{(u,v) \in H} (\pi(u) + \bar{\pi}(v))$ où $H = \{u \in U, v \in V / u \ll v \text{ ou } h_{(u,v)} \neq 0\}$. Donc il existe $u \in U$, $v \in V$ tel que :

- $g_{(u,v)} \neq 0$;
- $C'_{max}(g) = \pi(u) + \bar{\pi}(v)$;
- $h_{(u',v')} = 0$ pour tous les arcs (u', v') tel que $\pi(u') + \bar{\pi}(v') > C'_{max}(g)$.

La différence des flots g et h donne un cycle $\gamma = (u_0, v_0, u_1, v_1, \ldots, u_n = u_0)$ tel que :

- $u_0 = u$, $v_0 = v$;

– $\forall i \in [0 \ldots n-1]$, $g_{(u_i,v_i)} > h_{(u_i,v_i)}$ et $g_{(u_{i+1},v_i)} < h_{(u_{i+1},v_i)}$ (voir figure 2.9 pour $n = 3$).

Puisque $C'_{max}(h) < C'_{max}(g)$ on a $\pi(u_1) < \pi(u_0)$ et comme g satisfait la propriété P_{cross} on doit avoir $\bar{\pi}(v_1) \geq \bar{\pi}(v_0)$. De même pour tout $i \in [1 \ldots n-1]$ on a $\pi(u_i) < \pi(u_0)$ et la propriété P_{cross} sur g stipule qu'on doit avoir $\bar{\pi}(v_i) \geq \bar{\pi}(v_0)$. Pour $i = n-1$ on doit donc avoir $\bar{\pi}(v_{n-1}) \geq \bar{\pi}(v_0)$. Donc $C'_{max}(g) = \pi(u_0) + \bar{\pi}(v_0) \leq \pi(u_0) + \bar{\pi}(v_{n-1})$. Comme h est non nul sur l'arc (u_0, v_{n-1}) on en déduit que $C'_{max}(h)$ est supérieur ou égal à $C'_{max}(g)$, ce qui induit une contradiction avec l'hypothèse de départ. □

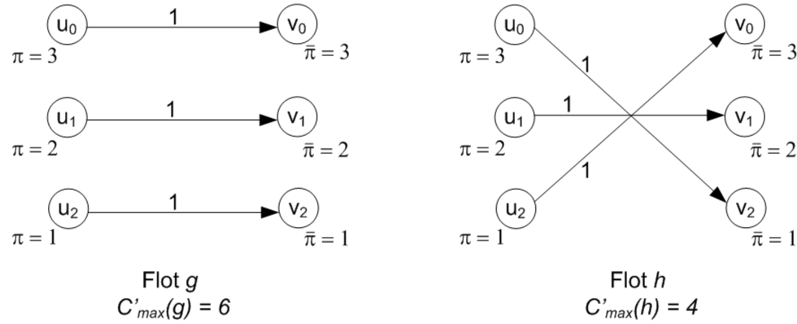


FIG. 2.9 – Le flot h vérifie P_{cross} alors que le flot g ne la vérifie pas

Algorithme 15 : Flot_Coupe : algorithme optimal pour le problème de la coupe

Entrées : U, V, h, out, in
Sorties : h

1 $U' \leftarrow \emptyset$ //ensemble des éléments de U utilisés par h ;
2 $V' \leftarrow \emptyset$ //ensemble des éléments de V utilisés par h ;
3 //choix de l'arc (i,j) :
4 $i \leftarrow \underset{u \in U-U'}{\operatorname{argmin}}(\pi(u))$;
5 $U' \leftarrow U' \cup \{i\}$;
6 $j \leftarrow \underset{v \in V-V'}{\operatorname{argmax}}(\bar{\pi}(v))$;
7 $V' \leftarrow V' \cup \{j\}$;
8 **Tant que** *il existe un élément u de U pour lequel* $out(u) \neq 0$ **faire**
9   //quantité de flot à mettre sur (i,j) :
10     $r \leftarrow \min(out(i), in(j))$;
11     $h_{(i,j)} \leftarrow r$;
12     $out(i) \leftarrow out(i) - r$;
13     $in(j) \leftarrow in(j) - r$;
14   //mise à jour de l'arc (i,j) :
15   **Si** $out(i) \neq 0$ **alors**
16         $j \leftarrow \underset{v \in V-V'}{\operatorname{argmax}}(\bar{\pi}(v))$;
17         $V' \leftarrow V' \cup \{j\}$;
18   **sinon**
19         $i \leftarrow \underset{u \in U-U'}{\operatorname{argmin}}(\pi(u))$;
20         $U' \leftarrow U' \cup \{i\}$;

Le problème de la coupe est résolu de manière optimale par l'algorithme `Flot_Coupe`. Pour cela, les quantités de flot dans la coupe (U, V) sont déterminées itérativement. A chaque itération, on choisit parmi les éléments u de U qui sont tels que $out(u) > 0$ celui de plus petit π et parmi les éléments v de V qui sont tels que $in(v) > 0$ celui de plus grand $\bar{\pi}$. L'arc (u, v) reçoit alors la plus grande quantité possible de flot en fonction de $out(u)$ et $in(v)$. De cette manière, on respecte la propriété P_{cross}. Le processus se termine lorsque tous les éléments de U et de V ont reçu suffisamment de flot.

Exemple 6. *Exemple de flot compatible avec une coupe.*

On considère une instance du RCPSP à 4 activités, 1 type de ressource en quantité 4. Toutes les activités consomment 2 unités de ressource. L'activité 1 dure 3 ; l'activité 2 dure 1 ; l'activité 3 dure 1 et l'activité 4 dure 2. La figure 2.10 illustre deux solutions du problème de la coupe avec une coupe (U, V) telle que $U = \{s, 1, 2\}$ et $V = \{2, 4, p\}$. Les valeurs out et in sont $out(s) = 0$; $out(1) = 2$; $out(2) = 2$; $in(3) = 2$; $in(4) = 0$; $in(p) = 2$. Le problème de la coupe est résolu de manière non optimale sur la figure (a) et de manière optimale sur la figure (b). Pour visualiser les dates de début des activités, on représente les diagrammes de Gantt associés à ces deux flots figure 2.11.

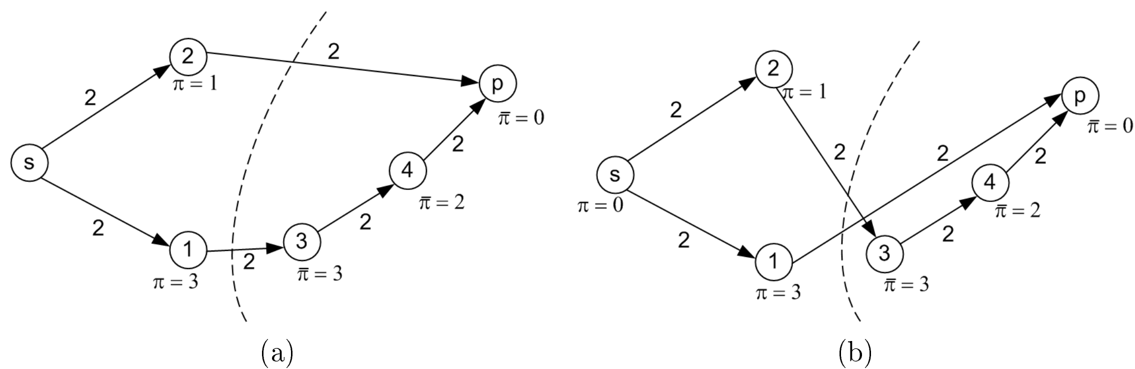


FIG. 2.10 – (a) Résolution non optimale ; (b) Résolution optimale

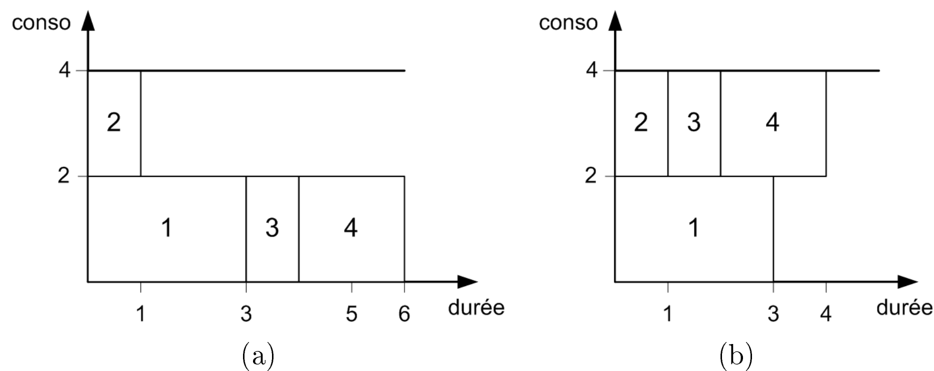


FIG. 2.11 – (a) Diagramme de Gantt associé au flot de la figure 2.10(a) ; (b) Diagramme de Gantt associé au flot de la figure 2.10(b)

2.2.2.3 Problème de l'insertion dans le cas d'un flot unique

On considère ici le cas où il n'y a qu'un type de ressources (donc un seul flot). Le cas où il y a plusieurs types de ressources (multiflot) est traité dans la section suivante.

Soit x_0 l'action à insérer de durée d_{x_0} et consommation ρ. Soit une coupe (U,V) telle que $\sum_{u\in U} out(u) = \sum_{v\in V} in(v) \geq \rho$. Un flot h est dit compatible avec l'insertion de x_0 dans (U,V) s'il vérifie la propriété (P_{insert}) suivante :

$$(P_{insert}) \begin{cases} \forall u \in U,\ out(u) = \sum_{v\in V\cup\{x_0\}} h_{(u,v)} \\ \forall v \in V,\ in(v) = \sum_{u\in U\cup\{x_0\}} h_{(u,v)} \\ \rho = \sum_{v\in V} h_{(x_0,v)} = \sum_{u\in U} h_{(u,x_0)} \end{cases}$$

On rappelle que $H = \{u \in U, v \in V/u \ll v \text{ ou } h_{(uv)} \neq 0\}$ et on pose de plus :
$H_2 = \{u \in U, v \in V/(u \ll x_0 \text{ ou } h_{(ux_0)} \neq 0) et (x_0 \ll v \text{ ou } h_{(x_0v)} \neq 0)\}$.
Pour un flot h qui vérifie P_{insert} on pose :
$C_1(h) = \max_{(u,v)\in H}(\pi(u) + \bar{\pi}(v))$;
$C_2(h) = \max_{(u,v)\in H_2}(\pi(u) + \bar{\pi}(v) + d_{x_0})$;
$C'_{max}(h) = max(C_1, C_2)$.

Définition du problème
Problème de l'insertion : *Etant donné un graphe biparti (X,E) tel que $X = U \cup V$ et une activité x_0 qui n'est pas dans X, trouver un flot compatible avec l'insertion de x_0 dans (U,V) qui minimise $C'_{max}(h)$.*

Solution du problème
Soit U_π l'ensemble des activités de U ordonnées dans le sens de leur π croissant. Pour traiter le problème de l'insertion, on commence par « raccrocher » x_0 à l'ensemble U puis on résout le problème de la coupe avec la coupe $(U \cup \{x_0\}, V)$. Pour raccrocher x_0 à U, il faut calculer la quantité de flot $h_{(u,x_0)}$, pour $u \in U$. Pour cela, on définit une *activité de raccrochage* $u_r \in U$ tel que x_0 prend du flot à u_r et aux actions immédiatement précédentes dans U_π.

Une activité de raccrochage $u_r \in U$ doit être telle que :

1. il n'existe pas d'activité $u \in U$ telle que $\pi(u_r) < \pi(u)$ et $u \ll x_0$
2. $\sum_{\{u\in U \ \pi(u)\leq\pi(u_r)\}} out(u) \geq \rho$.

L'insertion de x_0 dans la coupe (U,V) consiste alors à accrocher x_0 à partir de l'activité $u_r \in U$, comme décrit par l'algorithme 16. L'algorithme 16 calcule les quantités $h_{(u,x_0)}$, $u \in U$, de sorte que le flot h satisfait la propriété (P_{raccro}) suivante :

$$(P_{raccro}) \begin{cases} \forall u \in U, u' \in U,\ \left(\pi(u') < \pi(u) \leq \pi(u_r) \text{ et } h_{(u',x_0)} \neq 0\right) \Rightarrow h_{(u,x_0)} = out(u) \\ \forall u \in U,\ \pi(u) > \pi(u_r) \Rightarrow h_{(u,x_0)} = 0 \end{cases}$$

Ensuite, on calcule les flots dans la coupe $(u \cup \{x_0\}, V)$. Le processus complet est décrit par l'algorithme 17.

Algorithme 16 : Raccrochage

Entrées : U, x_0, u_r, ρ, h
Sorties : h

1 $U_\pi \leftarrow$ l'ensemble des éléments de U rangés dans l'ordre de leur π croissant ;
2 $u \leftarrow u_r$;
3 $out(x_0) \leftarrow 0$;
4 **Tant que** $out(x_0) \neq \rho$ **faire**
5     $r \leftarrow \min(out(u), \rho - out(x_0))$;
6     $h_{(u,x_0)} \leftarrow r$;
7     $out(x_0) \leftarrow out(x_0) + r$;
8     $out(u) \leftarrow out(u) - r$;
9     $i \leftarrow$ l'élément précédent dans U_π
10 $\pi(x_0) \leftarrow \pi(u_r) + d_{x_0}$;

Algorithme 17 : Insertion

Entrées : U, V, x_0, h
Sorties : h_{best}

1 $C_{best} \leftarrow +\infty$;
2 Calculer l'ensemble L_r des activités de raccrochage possibles pour x_0 dans U ;
3 **pour chaque** $u_r \in L_r$ **faire**
4     $h' \leftarrow h$;
5     $h =$ **Raccrochage**(U, x_0, u_r, ρ, h') ;
6     $h =$ **Flot_Coupe**($U \cup x_0$, V, h', out, in) ;
7   **Si** $C'_{max}(h') < C_{best}$ **alors**
8         $h_{best} \leftarrow h'$;
9         $C_{best} \leftarrow C'_{max}(h')$;

Théorème 2
L'algorithme 17 est valide pour le problème de l'insertion.

Démonstration

Pour prouver ce théorème, on montre que, si on considère g une solution au problème de l'insertion et si on pose u l'élément de plus grand π dans U, tel que $(u \ll x_0)$ ou $g_{(u,x_0)} > 0$, alors on peut alors modifier g de manière à obtenir la propriété (P_{raccro}) associée à u sans perdre la faisabilité de g et sans détériorer $C'_{max}(g)$.

Pour faire cela, on considère par exemple $x, x' \in A$ tels que :

$$\begin{cases} \pi(x) < \pi(x') \leq \pi(u) \\ g_{x',x_0} \neq 0 \text{ et } g_{x,x_0} \neq out(x) \end{cases}$$

On peut alors choisir $v \in V$ et rediriger le flot de la manière suivante :

$$\begin{aligned}
&\delta \leftarrow \min(g_{(x',x_0)}, g_{(x,v)});\\
&g_{(x',x_0)} \leftarrow g_{(x',x_0)} - \delta;\\
&g_{(x,v)} \leftarrow g_{(x,v)} - \delta;\\
&g_{(x',v)} \leftarrow g_{(x',v)} + \delta;\\
&g_{(x,x_0)} \leftarrow g_{(x,x_0)} + \delta;
\end{aligned}$$

En utilisant itérativement cette redirection du flot g vérifie (P_{raccro}) tout en restant solution du problème d'insertion.

Ainsi, soit g une solution optimale au problème de l'insertion, on peut le transformer en solution optimale g' qui satisfait (P_{raccro}) sans détériorer le makespan. Comme l'algorithme 17 balaie toutes les activités de raccrochage possibles, il y en a forcément une qui correspond à l'activité de raccrochage $u_{g'}$ associée au flot g'. De plus, d'après le théorème 1, l'algorithme `flot_coupe` construit un flot optimal pour la coupe $(U \cup \{x_0\}, V)$. Ainsi en utilisant `flot_coupe` après avoir raccroché x_0 à $u_{g'}$ on construit un flot optimal pour le problème de l'insertion. □

Exemple 7. *Insertion de l'activité 5 dans le graphe de la figure 2.10*
On reprend la coupe $U = \{s, 2, 1\}$, $V = \{3, 4, p\}$ de la figure 2.10, on veut insérer l'activité 5 de durée $d_5 = 2$ et consommation $\rho = 1$ dans cette coupe. L'activité de raccrochage choisie est l'activité 1. On a alors $h_{(1,5)} = 1$, les flots dans la coupe $(U \cup \{5\}, V)$ sont calculés de manière optimale par l'algorithme `Flot_coupe` qui donne $h_{(2,3)} = 2$, $h_{(1,p)} = 1$, $h_{(5,p)} = 1$ (voir figure 2.12(a)). Les coûts engendrés par cette insertion sont :

$C_1 = \max(\pi(1) + \bar{\pi}(p), \pi(2) + \bar{\pi}(3)) = 4$
$C_2 = \max(\pi(1) + \bar{\pi}(p) + d_5) = 5$
$C'_{max} \max(C_1, C_2) = 5$

Pour visualiser les dates de début des activités, on donne le diagramme de Gantt figure 2.12(b).

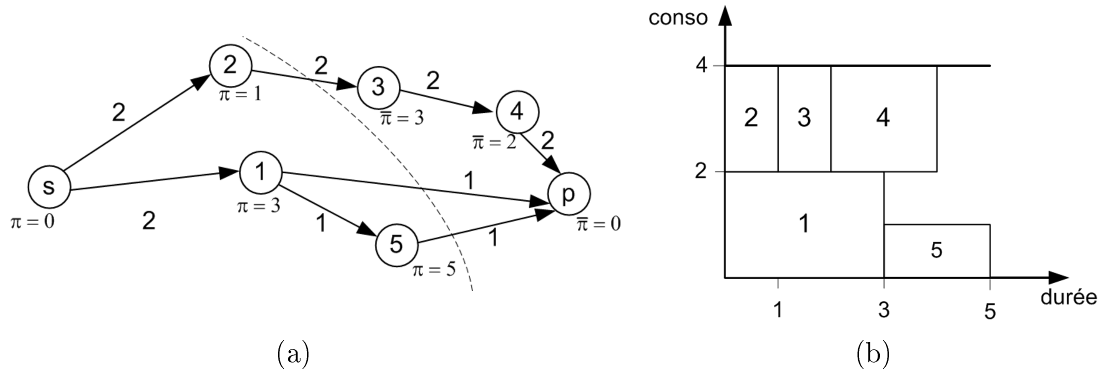


FIG. 2.12 – (a) Valeur du flot après insertion de l'activité 5 ; (b) Diagramme de Gantt associé au flot (a)

2.2.2.4 Problème de l'insertion multiflot

Dans le cas multiflot (plusieurs types de ressources), il suffit d'appliquer l'algorithme d'insertion à chacun des types de ressources et d'étendre la notion d'activité de raccrochage de la manière suivante. Une activité de raccrochage $u_r \in U$ est telle que :

1. il n'existe pas d'activité $u \in U$ telle que $\pi(u_r) < \pi(u)$ et $u \ll x_0$
2. $\forall k \in R, \quad \sum_{\substack{u \in U \\ \pi(u) \leq \pi(u_r)}} out(u) \geq r_{x_0 k}$.

On étend également le calcul du C'_{max} pour un multiflot $F = (f_1, \ldots, f_m)$ en posant : $C'_{max}(F) = \max_{k \in R} C'_{max}(f_k)$.

Enfin on note, pour une ressource $k \in R$, out_k le flot entrant dans la coupe et in_k le flot sortant de la coupe. On peut alors définir l'algorithme 18 d'insertion pour un multiflot.

Algorithme 18 : InsertionMulti

Entrées : U, V, x_0, $F = (f_1, \ldots, f_m)$
Sorties : F_{best}

1 $C_{best} \leftarrow +\infty$;
2 Calculer l'ensemble L_r des activités de raccrochage possibles pour x_0 dans U ;
3 **pour chaque** $i_r \in L_r$ **faire**
4     $F' \leftarrow F : \forall k \in R, f'_k \leftarrow f_k$;
5   **pour chaque** *type de ressource* $k \in R$ **faire**
6   f'_k = **Raccrochage**(U, x_0, i_r, $r_{x_0 k}$, f'_k) ;
7   f'_k = **Flot_Coupe**($U \cup x_0$, V, f'_k, out_k, in_k) ;
8   **Si** $C'_{max}(F') < C_{best}$ **alors**
9         $F_{best} \leftarrow F'$;
10         $C_{best} \leftarrow C'_{max}(F')$;

L'algorithme 18 est optimal pour le problème de l'insertion multiflot.

2.2.2.5 Notion d'ordre sur les arcs

Il est clair qu'il est intéressant de faire porter du flot aux arcs de $E_{\ll}$ puisque ces derniers sont forcément dans E et interviennent, par conséquent, dans le calcul du makespan. On évite, de cette manière, de créer trop d'arcs dans E_F et donc trop de contraintes liées aux échanges de ressources. En outre, les arcs les plus intéressants, en dehors des arcs de précédence, sont les arcs (u, v) de $U \times V$ tels que $\pi(u) + \bar{\pi}(v)$ est le plus petit possible. Il est donc intéressant d'ordonner les arcs suivant cette valeur. Pour cela, on définit σ un ordre sur les arcs de $U \times V - E_{\ll}$ tel qu'un arc (u, v) est plus petit qu'un arc (u', v') au sens de σ si :

$$\begin{cases} \quad \pi(u) + \bar{\pi}(v) < \pi(u') + \bar{\pi}(v') \\ \text{ou } \pi(u) + \bar{\pi}(v) = \pi(u') + \bar{\pi}(v') \text{ et } \pi(u) < \pi(u') \\ \text{ou } \pi(u) + \bar{\pi}(v) = \pi(u') + \bar{\pi}(v') \text{ et } \pi(u) = \pi(u') \text{ et } \bar{\pi}(v) > \bar{\pi}(v') \end{cases}$$

On note E_σ les arcs de $U \times V - E_{\ll}$ ordonnées suivant l'ordre σ. L'idée est alors de

faire porter du flot en priorité aux arcs de $E_{\ll}$ puis aux arcs les plus petits de E_{σ} lorsqu'on résout le problème de la coupe.

2.2.2.6 Calcul de la coupe

On connait maintenant la technique pour insérer un élément x_0 dans une coupe (U, V) de l'ensemble X. Il reste à définir comment on choisit une telle coupe. On rappelle qu'une coupe compatible avec x_0 doit avoir les propriétés suivantes :

- pour tout $u \in U$, $x \in X$, si $x \ll u$ ou $F_{(x,u)} \neq 0$, alors $x \in U$;
- si un élément $x \in X$ est tel que $x \ll x_0$ alors x appartient à U ;
- si un élément $x \in X$ est tel que $x_0 \ll x$ alors x appartient à V ;

Théorème 3
Le problème qui consiste à trouver la meilleure coupe pour l'insertion de x_0 est NP-complet.

Démonstration
On considère la configuration suivante dans laquelle on souhaite insérer une nouvelle activité x_0 dans le graphe partiel $G' = (X', E')$:

- une seule ressource disponible en quantité M, chaque activité i consomme r_i ;
- aucune contrainte de précédence ;
- $\sum_{i \in X'} r_i = M$;
- $r_{x_0} = M/2$;
- les activités i de X' ont une durée 1 ;
- x_0 a une durée 2.

Le flot F est alors trivialement défini par : pour tout i dans X', $F_{(s,i)} = r_i = F_{(i,p)}$ et le makespan est égal à 1 (voir figure 2.13 (a) et (b)). Il est clair que dans le meilleur des cas l'insertion de x_0 fait passer le makespan à 2 (voir figure 2.13(c)). Dans cet exemple, déterminer la meilleure coupe est un problème de 2-partition, qui est NP-complet. □

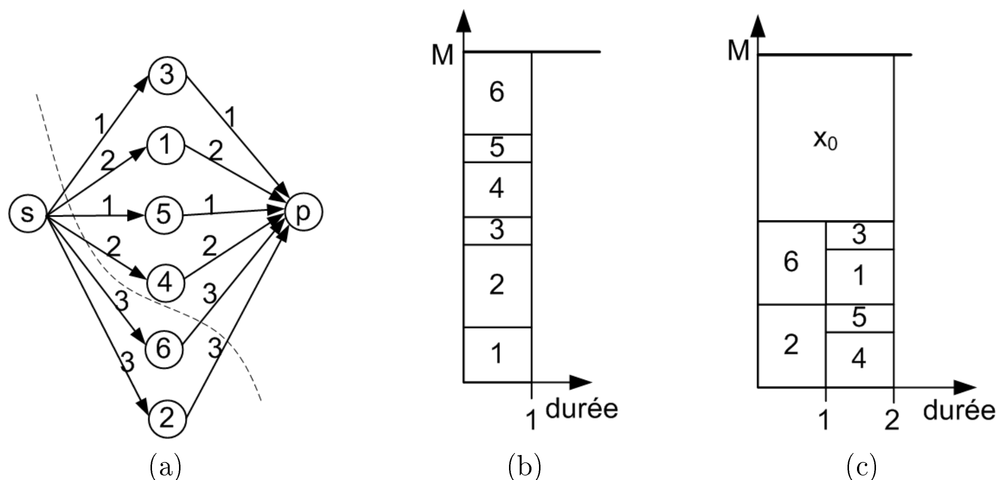


FIG. 2.13 – (a) flot initial, la coupe optimale pour l'insertion de x_0 est représentée en pointillé ; (b) Gantt associé au flot (a) ; (c) Gantt après insertion de x_0

Comme trouver la meilleure coupe peut être très coûteux, on se contente de calculer les coupes compatibles avec x_0 de la manière suivante :

- on calcule les dates t auxquelles peut commencer x_0 ;
- pour chaque date t, on définit la coupe (U, V) telle que les activités qui terminent au plus tard en t sont dans U, les autres sont dans V.

2.2.2.7 Algorithme de résolution du RCPSP

Les techniques et algorithmes définis précédemment permettent de construire l'algorithme 19 de résolution complète du problème de multiflot associé à une instance du RCPSP. Notons que, comme la procédure `InsertionMulti` recalcule entièrement les flots dans la coupe, il est nécessaire de supprimer tout le flot initial dans la coupe avant de l'utiliser.

Algorithme 19 : Evaluer (version détaillée)

Entrées : $A = \{1, \ldots, n\}$, λ (séquence des activités dans un ordre aléatoire)
Sorties : $F = (f_1 \ldots f_m)$

```
1  X ← {s, p} // initialisation des sommets de G' ;
2  ∀k ∈ R, f_k(sp) ← M_k // initialisation du multiflot ;
3  Tant que il reste des activités non insérées faire
4  |  Choisir la prochaine activité x0 dans λ ;
5  |  Calculer les dates t auxquelles peut commencer x0 ;
6  |  Calculer les coupes (U, V) en fonction de t ;
7  |  C_best ← +∞ ;
8  |  pour chaque coupe (U, V) faire
9  |  |  F' ← F ;
10 |  |  Supprimer dans F' les flots relatifs à la coupe ;
11 |  |  F' ← InsertionMulti(U, V, x0, F') ;
12 |  |  Si C'_max(F') < C_best alors
13 |  |  |  F_best ← F' ;
14 |  |  |  C_best ← C'_max(F')
15 |  F ← F_best ;
```

2.2.3 Recherche locale

Les procédures d'insertion, décrites dans les sections précédentes, sont intéressantes car elles nous fournissent non seulement une technique d'insertion pour construire une solution mais peuvent aussi être utilisées comme opérateur de recherche locale. En effet, on peut essayer d'améliorer une solution décrite par un graphe $G = (X, E)$ et un multiflot $F = (f_1 \ldots f_m)$ en supprimant une (ou plusieurs) activité de X et en la (ou les) réinsérant au mieux. Il est clair que la suppression d'une activité x_0 modifie le multiflot F. Le multiflot F est modifié en considérant une coupe compatible avec la suppression de x_0 et en utilisant l'algorithme `flot_coupe` qui reconstruit de façon optimale le flot dans cette coupe. L'activité x_0 est ensuite réinsérée dans le graphe via le procédé décrit par la boucle principale de l'algorithme 19. Le processus d'extraction d'une activité est décrit par l'algorithme 20.

Algorithme 20 : Suppression

Entrées : x_0

1 **Construire** la coupe (U, V) : U est l'ensemble des activités qui donnent du flot à x_0 et V est l'ensemble des activités qui prennent du flot de x_0 ;

2 $\forall k \in R,\ \forall u \in U,\ out_k(u) = \sum_{x \in V \cup x_0} f_{k(u,x)}$;

3 $\forall k \in R,\ \forall v \in V,\ in_k(v) = \sum_{x \in U \cup x_0} f_{k(x,v)}$;

4 Supprimer dans F les flots relatifs à la coupe ;

5 **pour chaque** *type de ressources* $k \in R$ **faire**

6 $\quad f_k =$ **Flot_Coupe**$(U,\ V,\ f_k,\ out_k,\ in_k)$;

On peut alors définir un algorithme de transformation locale qui supprime un ensemble d'activités W d'une solution et les réinsère. L'algorithme 21 donne le schéma général de ce processus : les activité de W sont tout d'abord supprimées une à une grâce à la procédure `Suppression`, elles sont ensuite insérées de la même manière que dans l'algorithme `Evaluer` (algorithme 19), comme si elles n'avaient jamais été dans l'ordonnancement.

Algorithme 21 : Transformation

Entrées : W, F

Sorties : F

1 **pour chaque** $w \in W$ **faire**

2 $\quad$ **Suppression**(w);

3 **pour chaque** $w \in W$ **faire**

4 $\quad$ Réinsérer w et mettre à jour F tel que décrit par la boucle principale de l'algorithme 19 ;

L'algorithme transformation nous permet de construire une recherche locale. Pour tester l'efficacité de cet opérateur, on a conçu deux recherches locales : une descente et un *random walk* (voir chapitre 1 pour la description de ces algorithmes). La solution initiale est générée par l'algorithme `Evaluer` (algorithme 19) et l'opérateur de transformation est donné par l'algorithme `Transformation` (algorithme 21). Pour définir complètement la recherche locale, il reste à calculer l'ensemble passé en paramètre de `Transformation`. Plusieurs stratégies ont été testées :

« activité unique » : l'ensemble W est formé d'une seule activité choisie aléatoirement ;

« date » : une date t est choisie aléatoirement entre 0 et C_{max}, l'ensemble W est formé des activités actives en t ;

« chemin critique » : l'ensemble W est formé des activités du chemin critique ;

L'algorithme 22 donne le schéma général de la recherche locale (pour la descente). On note $C_{max}(F)$ le makespan d'une solution du RCPSP associée au multiflot F.

Algorithme 22 : RechercheLocale (descente)

Entrées :
$nbTransfo$ //nombre de transformations,
S //stratégie choisie,
F //multiflot associé à une solution initiale
Sorties : F

1 $cpt \leftarrow 0$;
2 **Tant que** $cpt < nbTransfo$ **faire**
3   **Calculer**, avec la stratégie S, un sous ensemble W de X ;
4   $F' \leftarrow$ **Transformation**(W, F) ;
5   **Si** $C_{max}(F') < C_{max}(F)$ **alors**
6     $F \leftarrow F'$;

2.2.4 Résultats numériques

On présente les résultats des tests pour l'algorithme `Evaluer` puis pour les deux recherches locales avec les différentes stratégies de choix pour W envisagées. Les tests ont été effectués sur un PC AMD Opteron, 2.1GHz, sous Linux. Les algorithmes sont programmés en C++. Les instances utilisées sont celles de la PSPLIB à 30, 60 et 120 activités (disponibles sur le site `http://129.187.106.231/psplib/`, 480 instances à 30 et 60 activités et 600 instances à 120 activités).

2.2.4.1 Performance de l'heuristique d'insertion

Le tableau 2.3 donne les résultats de l'algorithme `Evaluer`. Comme cet algorithme est non déterministe on l'exécute plusieurs fois et on garde la meilleure solution obtenue. Les résultats sont donnés pour 100, 1 000, 5 000 et 50 000 exécutions. On utilise les notations :
- |A| : nombre d'activités ;
- |R| : nombre de types de ressources ;
- rep. : nombre de fois où l'algorithme est exécuté (= nombre de séquences λ évaluées) ;
- cpu : temps cpu moyen pour la totalité des exécutions d'une instance (en seconde) ;
- gap : écart en pourcentage entre la meilleure solution trouvée par notre méthode et la solution de référence.

Dans le tableau 2.3 la solution de référence est, pour les instances à 30 activités, la solution optimale. Pour les instances à 60 et 120 activités la solution de référence est la borne inférieure triviale : makespan obtenu en relaxant les contraintes de ressources.

Il est clair que l'heuristique `Evaluer` est plus performante si on lui laisse le temps d'évaluer plus d'ordonnancements. Cependant, on a constaté que pour certaines instances elle trouve la solution optimale très facilement (systématiquement dès les premières évaluations) alors que, pour d'autres, la solution optimale n'est pas atteinte même au bout de 50 000 évaluations.

Pour expliquer ce phénomène, nous avons réalisé une étude statistique des instances à 30 activités de la PSPLIB, pour lesquelles on connaît les valeurs des solutions optimales. Cette étude est synthétisée à travers le tableau 2.4 qui donne les caractéristiques des instances pour les 48 groupes (de j301 à j3048) ainsi que les résultats obtenus par 1 000 exécutions

|A|	|R|	rep.	cpu (s)	gap (%)
30	4	100	0.6	1.87
30	4	1 000	6.3	0.92
30	4	5 000	31.6	0.53
30	4	50 000	317.4	0.28
60	4	100	4.5	16.91
60	4	1 000	53	15.37
60	4	5 000	243	14.56
60	4	50 000	2432	13.77
120	4	100	29.6	52.33
120	4	1 000	515	48.84
120	4	5 000	2608	47.05
120	4	50 000	25961	45.07

TAB. 2.3 – Tests `Evaluer`, de 100 à 50 000 runs, le gap est exprimé par rapport à la solution optimale dans le cas de 30 jobs et par rapport au makespan trivial pour les autres

de notre heuristique pour chaque groupe.

Dans le tableau 2.4, on utilise les notations suivantes :

- BI triviale : makespan obtenu en relaxant les contraintes de ressources (makespan induit par les contraintes de précédence uniquement) ;
- nb type : nombre moyen de types de ressources différents consommés par les activités d'une instance ;
- paral. moyen : nombre moyen d'activités en parallèle dans la solution optimale pour une instance ;
- paral. max : nombre maximal d'activités en parallèle dans la solution optimale pour une instance ;
- cpu : temps cpu pour la totalité des exécutions (en seconde) ;
- gap : écart en pourcentage entre la meilleure solution trouvée par notre méthode et la solution optimale.

Pour la borne inférieure (BI) triviale et le nombre de job en parallèle la colonne est divisée en 3 : on donne le minimum, le maximum et la moyenne des instances pour le groupe considéré. Le nombre moyen de types de ressources différents consommés par les activités d'une instance est systématiquement le même pour chaque instance du même groupe.

On constate, à travers cette étude, que les 480 instances à 30 jobs sont extrêmement biaisées : le makespan optimal correspond très souvent au makespan trivial (216 instances sur 480). Il n'y a que 4 valeurs différentes pour le nombre moyen de types de ressources consommés pour une instance, les contraintes de ressources et de précédence sont telles que peu d'activités peuvent être traitées en parallèle (seulement entre 2 et 4 en moyenne et rarement plus de 6). On constate également que notre heuristique fournit de meilleurs résultats pour les instances dont le makespan optimal est proche du makespan trivial, cela s'explique par le fait qu'elle est guidée, lors de l'insertion d'une activité, par les contraintes de précédence.

	BI triviale			nb type	paral. moyen			paral. max			cpu(s)	gap (%)
	min	max	moy		min	max	moy	min	max	moy		
j301	0.0	26.3	13.8	1.0	2.4	3.4	3.0	4.0	7.0	5.4	5.8	0.0
j302	0.0	11.8	4.4	1.0	3.0	3.9	3.5	5.0	7.0	6.0	6.3	0.0
j303	0.0	15.2	3.1	1.0	2.0	3.6	2.9	4.0	6.0	5.5	6.3	0.0
j304	0.0	0.0	0.0	1.0	2.8	3.9	3.2	5.0	7.0	5.9	6.0	0.0
j305	25.0	76.7	38.7	2.03	2.0	3.0	2.4	3.0	5.0	4.4	5.8	2.7
j306	0.0	17.5	7.6	2.03	2.7	3.8	3.2	4.0	7.0	5.4	6.6	0.6
j307	0.0	14.0	3.8	2.03	2.5	3.9	3.4	5.0	7.0	5.8	6.7	0.0
j308	0.0	0.0	0.0	2.03	2.5	4.3	3.4	4.0	8.0	6.5	7.4	0.0
j309	25.5	104.0	52.3	3.03	1.6	2.3	2.0	3.0	4.0	3.4	5.6	4.7
j3010	0.0	10.8	5.4	3.03	2.8	3.8	3.2	4.0	6.0	4.7	6.9	1.6
j3011	0.0	5.0	1.4	3.03	2.3	4.3	3.2	4.0	7.0	5.1	7.0	1.0
j3012	0.0	0.0	0.0	3.03	3.0	4.7	3.7	4.0	8.0	6.1	8.0	0.0
j3013	31.1	121.0	61.2	4.0	1.6	2.3	2.0	2.0	3.0	2.9	5.7	6.4
j3014	0.0	20.5	7.1	4.0	2.7	3.7	2.9	3.0	5.0	4.0	6.7	3.5
j3015	0.0	8.7	1.2	4.0	2.6	3.7	3.0	4.0	6.0	4.6	7.4	0.5
j3016	0.0	0.0	0.0	4.0	3.0	4.5	3.5	4.0	7.0	5.9	8.6	0.0
j3017	0.0	42.2	14.7	1.0	2.5	3.3	2.8	4.0	7.0	5.2	5.5	0.0
j3018	4.0	19.5	8.7	1.0	2.8	3.3	3.0	5.0	6.0	5.4	5.7	0.0
j3019	0.0	11.4	2.8	1.0	2.5	4.1	3.3	4.0	7.0	6.0	5.9	0.0
j3020	0.0	0.0	0.0	1.0	2.4	3.9	3.4	4.0	7.0	5.6	6.2	0.0
j3021	22.6	60.5	35.3	2.03	1.8	2.6	2.2	3.0	4.0	3.8	5.2	0.3
j3022	3.3	15.6	7.9	2.03	2.3	3.6	3.0	4.0	7.0	5.2	6.4	0.0
j3023	0.0	8.6	2.6	2.03	2.5	3.3	2.9	4.0	7.0	5.3	6.5	0.0
j3024	0.0	0.0	0.0	2.03	2.5	4.1	3.2	5.0	7.0	5.7	6.9	0.0
j3025	23.4	81.6	59.9	3.03	1.7	2.4	2.0	3.0	3.0	3.0	5.3	4.1
j3026	0.0	15.6	3.8	3.0	2.2	3.7	2.7	3.0	5.0	4.1	6.4	1.1
j3027	0.0	6.5	0.6	3.03	2.4	3.5	2.9	4.0	5.0	4.5	6.9	0.2
j3028	0.0	0.0	0.0	3.03	2.4	3.5	3.0	4.0	6.0	5.0	7.4	0.0
j3029	37.1	114.0	73.8	4.0	1.7	2.2	1.9	2.0	3.0	2.5	5.2	3.7
j3030	3.8	23.3	10.1	4.0	2.5	3.7	2.8	3.0	5.0	4.0	6.2	3.6
j3031	0.0	12.2	2.9	4.0	2.6	3.4	2.9	3.0	5.0	4.5	7.1	0.9
j3032	0.0	0.0	0.0	4.0	2.7	4.1	3.2	5.0	8.0	5.5	7.9	0.0
j3033	0.0	31.0	14.6	1.0	2.3	3.2	2.8	3.0	6.0	4.7	5.2	0.0
j3034	0.0	16.0	5.3	1.0	2.4	3.5	2.8	4.0	6.0	4.6	5.2	0.0
j3035	0.0	7.3	2.0	1.0	2.5	3.2	2.9	4.0	6.0	5.3	5.6	0.0
j3036	0.0	0.0	0.0	1.0	2.5	3.6	2.9	5.0	7.0	5.5	5.8	0.0
j3037	23.9	71.7	43.9	2.0	1.7	2.3	2.1	3.0	4.0	3.8	5.1	0.1
j3038	0.0	15.1	8.0	2.03	2.3	3.16	2.7	3.0	5.0	4.1	5.9	0.0
j3039	0.0	15.0	4.5	2.03	2.3	3.4	2.9	4.0	5.0	4.7	6.2	0.0
j3040	0.0	0.0	0.0	2.03	2.7	3.8	3.1	4.0	6.0	5.3	6.5	0.0
j3041	33.3	81.6	60.5	3.0	1.4	2.0	1.8	2.0	4.0	3.2	4.8	1.9
j3042	0.0	22.4	7.4	3.03	2.0	3.1	2.6	3.0	5.0	3.9	6.0	1.0
j3043	0.0	3.8	1.8	3.03	2.3	3.1	2.7	4.0	5.0	4.2	6.4	0.8
j3044	0.0	0.0	0.0	3.03	2.5	3.7	3.2	4.0	7.0	5.3	7.3	0.0
j3045	40.3	98.4	59.4	4.0	1.3	1.9	1.6	2.0	3.0	2.3	4.5	2.0
j3046	1.6	25.5	8.1	4.0	2.4	3.0	2.7	3.0	4.0	3.6	6.0	2.9
j3047	0.0	10.0	3.0	4.0	2.7	3.5	3.0	4.0	5.0	4.2	6.8	0.6
j3048	0.0	0.0	0.0	4.0	2.3	3.5	3.0	5.0	6.0	5.3	7.9	0.0

TAB. 2.4 – Performances de l'heuristique par rapport aux caractéristiques des instances

2.2.4.2 Performance des recherches locales implémentées dans le cadre d'un GRASP

Pour tester l'efficacité des transformations locales que nous proposons, nous les avons utilisées au travers d'un GRASP. Nous avons utilisé des recherches locales basiques (descente et *random walk*). L'algorithme d'évaluation fournit la solution initiale S_{init}, on effectue ensuite la recherche locale qui applique 100 transformations au maximum. Le processus est répété N fois.

Le tableau 2.5 synthétise les résultats obtenus. La colonne 1 donne la stratégie utilisée : d'une part, on choisit pour l'ensemble des activités à replacer W, une stratégie parmi les trois suivantes.

- « date »: une date t est choisie aléatoirement entre 0 et C_{max}, l'ensemble W est formé des activités actives en t ;
- « critique »: l'ensemble W est formé des activités du chemin critique ;
- « job »: l'ensemble W est formé d'une seule activité choisie aléatoirement.

D'autre part, on choisit le type de recherche locale : descente (Desc.) ou *random walk* (Walk). La deuxième colonne donne le nombre N de réplications. Les colonnes suivantes donnent les résultats pour les instance à 30, 60 et 120 activités, elles sont divisées en trois parties :

- « gap(%) »: pourcentage d'écart entre le coût de la meilleure solution trouvée S et de la solution de référence S_{ref} (S_{ref} est la solution optimale pour les instances à 30 activités et le makespan trivial pour les instances à 60 et 120 activités). Le *gap* est donc égal à $\frac{S-S_{ref}}{S_{ref}} \times 100$;
- « cpu(s) »: temps total d'exécution (en seconde) ;
- « up(%) »: pourcentage d'amélioration apporté par la recherche locale. On a donc $up = \frac{S_{init}-S}{S} \times 100$)

strategie	N	j30			j60			j120		
		gap(%)	cpu(s)	up(%)	gap(%)	cpu(s)	up(%)	gap(%)	cpu(s)	up(%)
date + Desc.	10	0.36	1.64	10.45	3.82	9.33	11.17	12.41	63.73	16.72
date + Desc.	50	0.12	8.42	10.68	3.34	46.93	11.25	11.34	321.13	17.04
date + Walk	10	0.69	1.60	10.14	14.10	9.17	10.21	44.30	61.99	14.50
date + Walk	50	0.37	8.19	10.32	13.47	46.10	10.27	42.77	312.63	14.77
critique + Desc.	10	0.85	3.77	10.07	14.34	20.11	9.16	45.41	210.80	13.11
critique + Desc.	50	0.49	19.41	9.85	13.69	101.50	9.60	43.71	1055.90	13.04
critique + Walk	10	0.91	4.18	10.41	14.63	22.55	9.57	45.61	227.11	14.15
critique + Walk	50	0.56	21.50	10.20	13.97	113.60	9.83	44.17	1139.40	13.99
job + Desc.	10	0.72	0.57	9.55	14.30	2.43	8.69	46.46	15.64	11.21
job + Desc.	50	0.30	2.88	9.33	13.53	12.23	9.09	44.13	79.32	11.23
job + Walk	10	0.78	0.57	9.60	14.59	2.42	8.50	47.42	15.20	10.60
job + Walk	50	0.37	2.88	9.38	13.73	12.18	8.92	45.13	78.66	10.68

TAB. 2.5 – Résultats obtenus par le GRASP

Le tableau 2.5, montre d'une part, que la descente est une stratégie globalement plus efficace que le *random walk*, et d'autre part, parmi les trois stratégies de sélection des activités à replacer, celle qui s'avère être la plus efficace est le choix suivant une date

aléatoire. On constate également que l'amélioration apportée par la recherche locale est plus importante sur les problèmes à 120 activités, probablement parce que l'heuristique de construction donne des résultats de moins bonne qualité sur ces instances et qu'il est donc plus facile de les améliorer.

2.2.4.3 Comparaison des performances avec les méthodes existantes

Même si notre objectif initial n'est pas de concevoir des méthodes dédiées au RCPSP classique, on propose ici de situer l'efficacité de notre algorithme par rapport au classement proposé par Kolisch et Hartmann [KH06] des meilleures heuristiques et métaheuristiques connues pour le RCPSP. L'approche la plus couramment utilisée pour observer les performances des méthodes approchées est de donner, pour un nombre d'évaluation d'ordonnancements fixé, l'écart du meilleur makespan trouvé par l'heuristique par rapport à la solution optimale (ou à une borne inférieure).

Dans le classement de [KH06] sont répertoriés 28 algorithmes classés selon leur écart à la solution optimale ou à la borne inférieure triviale pour 50 000 évaluations d'ordonnancement. Les résultats obtenus pour 1 000 et 5 000 évaluations sont également répertoriés. Il est clair que notre schéma heuristique ne met pas en jeu de méthodes complexes d'amélioration pour obtenir des résultats qui concurrencent les meilleurs algorithmes, le but étant de concevoir une méthode facilement adaptable à des extensions du RCPSP et non d'apporter de nouvelles solutions optimales.

Pour réaliser ces comparaisons, nous avons choisi la stratégie qui nous fournit les meilleurs résultats dans le tableau 2.5, à savoir la descente et le choix des activités à supprimer/réinsérer suivant une date aléatoire. Le nombre maximal d'ordonnancements évalués par le GRASP est donné par $nbTransfo \times N$ où $nbTransfo$ est le nombre de transformations autorisées dans la recherche locale et N le nombre de réplications. Le paramétrage qui donne les résultats référencés dans les tableaux comparatifs est présenté tableau 2.6.

	j30			j60			j120		
	1 000	5 000	50 000	1 000	5 000	50 000	1 000	5 000	50 000
N	10	50	100	5	10	100	2	2	25
$nbTransfo$	100	100	500	200	500	500	500	2 500	2 000

Tab. 2.6 – Paramétrage du GRASP en fonction du nombre d'activités et du nombre total d'évaluations autorisées (1 000, 5 000 et 50 000)

Les abréviations suivantes sont utilisées dans les tableaux 2.7 à 2.9 :
GA : algorithme génétique ;
FBI : amélioration forward-backward
LFT : « latest finish time » (règle de priorité)
TS : recherche tabou
WCS : « worst case slack » (règle de priorité)

Algorithme	Référence	nb max. ordo.		
		1 000	5 000	50 000
GA, TS - path relinking	Kochetov, Stolyar [KS03]	0.10	0.04	0.00
Scatter Search - FBI	Debels et al. [DRLV06]	0.27	0.11	0.01
GA - hybrid, FBI	Valls et al. [VBQ03]	0.27	0.06	0.02
GA - FBI	Valls et al. [VBQ05]	0.34	0.20	0.02
GA - forw.-backw., FBI	Alcaraz et al. [AMR04]	0.25	0.06	0.03
GRASP	**cette thèse**	**0.36**	**0.12**	**0.03**
GA - forw.-backward	Alcaraz, Maroto [AM01]	0.33	0.12	-
sampling - LFT, FBI	Tormos, Lova [TL03b]	0.25	0.13	0.05
TS - activity list	Nonobe, Ibaraki [NI02]	0.46	0.16	0.05
sampling - LFT, FBI	Tormos, Lova [TL01]	0.30	0.16	0.07
GA - self-adapting	Hartmann [Har02]	0.38	0.22	0.08
GA - activity list	Hartmann [Har98]	0.54	0.25	0.08
sampling - LFT, FBI	Tormos, Lova [TL03a]	0.30	0.17	0.09
TS - activity list	Klein [TL00]	0.42	0.17	-
sampling - random, FBI	Valls et al. [VBQ05]	0.46	0.28	0.11
SA - activity list	Bouleimen, Lecocq [BL03b]	0.38	0.23	-
GA - late join	Coelho, Tavares [CT03]	0.74	0.33	0.16
sampling - adaptive	Schirmer [Sch00]	0.65	0.44	-
TS - schedule scheme	Baar et al. [BBK98]	0.86	0.44	-
sampling - adaptive	Kolisch, Drexl [KD96]	0.74	0.52	-
GA - random key	Hartmann [Har98]	1.03	0.56	0.23
sampling - LFT	Kolisch [Kol96b]	0.83	0.53	0.27
sampling - global	Coelho, Tavares [CT03]	0.81	0.54	0.28
sampling - random	Kolisch [Kol95]	1.44	1.00	0.51
GA - priority rule	Hartmann [Har98]	1.38	1.12	0.88
sampling - WCS	Kolisch [Kol96b, Kol96a]	1.40	1.28	-
sampling - LFT	Kolisch [Kol96b]	1.40	1.29	1.13
sampling - random	Kolisch [Kol95]	1.77	1.48	1.22
GA - problem space	Leon, Ramamoorthy [LR95]	2.08	1.59	-

Tab. 2.7 – Ecart moyen à la solution optimale pour les instances de la PSPLIB de type j30 (tableau original extrait de [KH06])

On constate que notre heuristique a une efficacité très satisfaisante par rapport aux autres méthodes sachant que nous nous comparons à des métaheuristiques sophistiquées (algorithmes génétiques, colonies de fourmis, *scatter search* ...) alors que notre méthode utilise une métaheuristique basique. La méthode GRASP est la sixième méthode sur 29, et elle fait partie des meilleures méthodes publiées pour les instances à 30 jobs.

Algorithme	Référence	nb max. ordo.		
		1 000	5 000	50 000
Scatter Search - FBI	Debels et al. [DRLV06]	11.73	11.10	10.71
GA - hybrid, FBI	Valls et al. [VBQ03]	11.56	11.10	10.73
GA, TS - path relinking	Kochetov, Stolyar [KS03]	11.71	11.17	10.74
GA - FBI	Valls et al. [VBQ05]	12.21	11.27	10.74
GA - forw.-backw., FBI	Alcaraz et al. [AMR04]	11.89	11.19	10.84
GA - self-adapting	Hartmann [Har02]	12.21	11.70	11.21
GA - activity list	Hartmann [Har98]	12.68	11.89	11.23
sampling - LFT, FBI	Tormos, Lova [TL03b]	11.88	11.62	11.36
GRASP	**cette thèse**	**12.70**	**11.93**	**11.40**
sampling - LFT, FBI	Tormos, Lova [TL03a]	12.14	11.82	11.47
GA - forw.-backward	Alcaraz, Maroto [AM01]	12.57	11.86	-
sampling - LFT, FBI	Tormos, Lova [TL01]	12.18	11.87	11.54
SA - activity list	Bouleimen, Lecocq [BL03b]	12.75	11.90	-
TS - activity list	Klein [TL00]	12.77	12.03	-
TS - activity list	Nonobe, Ibaraki [NI02]	12.97	12.18	11.58
sampling - random, FBI	Valls et al. [VBQ05]	12.73	12.35	11.94
sampling - adaptive	Schirmer [Sch00]	12.94	12.58	-
GA - late join	Coelho, Tavares [CT03]	13.28	12.63	11.94
GA - random key	Hartmann [Har98]	14.68	13.32	12.25
GA - priority rule	Hartmann [Har98]	13.30	12.74	12.26
sampling - adaptive	Kolisch, Drexl [KD96]	13.51	13.06	-
sampling - WCS	Kolisch [Kol96b, Kol96a]	13.66	13.21	-
sampling - global	Coelho, Tavares [CT03]	13.80	13.31	12.83
sampling - LFT	Kolisch [Kol96b]	13.59	13.23	12.85
TS - schedule scheme	Baar et al. [BBK98]	13.80	13.48	-
GA - problem space	Leon, Ramamoorthy [LR95]	14.33	13.49	-
sampling - LFT	Kolisch [Kol96b]	13.96	13.53	12.97
sampling - random	Kolisch [Kol95]	14.89	14.30	13.66
sampling - random	Kolisch [Kol95]	15.94	15.17	14.22

Tab. 2.8 – Ecart moyen au makespan trivial pour les instances de la PSPLIB de type j60 (tableau original extrait de [KH06])

On constate, à travers le tableau 2.8, que les meilleures méthodes sur les instances à 30 jobs ne sont pas forcément les meilleures sur les méthodes à 60 jobs. Ainsi, par exemple, l'algorithme de [KS03] passe de la première à la troisième place et celui de [DRLV06] passe à la première place alors qu'il était en deuxième position. Le GRASP est à la neuvième place, il reste dans le haut du classement.

Algorithme	Référence	nb max. ordo.		
		1 000	5 000	50 000
GA - hybrid, FBI	Valls et al. [VBQ03]	34.07	32.54	31.24
GA - forw.-backw., FBI	Alcaraz et al. [AMR04]	36.53	33.91	31.49
Scatter Search - FBI	Debels et al. [DRLV06]	35.22	33.10	31.57
GA - FBI	Valls et al. [VBQ05]	35.39	33.24	31.58
GA, TS - path relinking	Kochetov, Stolyar [KS03]	34.74	33.36	32.06
population-based - FBI	Valls et al. [VBQ05]	35.18	34.02	32.81
GA - self-adapting	Hartmann [Har02]	37.19	35.39	33.21
sampling - LFT, FBI	Tormos, Lova [TL03b]	35.01	34.41	33.71
ant system	Merkle et al. [MMS02]	-	35.43	-
GA - activity list	Hartmann [Har98]	39.37	36.74	34.03
sampling - LFT, FBI	Tormos, Lova [TL03a]	36.24	35.56	34.77
GRASP	**cette thèse**	**38.67**	**36.60**	**35.00**
sampling - LFT, FBI	Tormos, Lova [TL01]	36.49	35.81	35.01
GA - forw.-backward	Alcaraz, Maroto [AM01]	39.36	36.57	-
TS - activity list	Nonobe, Ibaraki [NI02]	40.86	37.88	35.85
GA - late join	Coelho, Tavares [CT03]	39.97	38.41	36.44
sampling - random, FBI	Valls et al. [VBQ05]	38.21	37.47	36.46
SA - activity list	Bouleimen, Lecocq [BL03b]	42.81	37.68	-
GA - priority rule	Hartmann [Har98]	39.93	38.49	36.51
sampling - adaptive	Schirmer [Sch00]	39.85	38.70	-
sampling - LFT	Kolisch [Kol96b]	39.60	38.75	37.74
sampling - WCS	Kolisch [Kol96b, Kol96a]	39.65	38.77	-
GA - random key	Hartmann [Har98]	45.82	42.25	38.83
sampling - adaptive	Kolisch, Drexl [KD96]	41.37	40.45	-
sampling - global	Coelho, Tavares [CT03]	41.36	40.46	39.41
GA - problem space	Leon, Ramamoorthy [LR95]	42.91	40.69	-
sampling - LFT	Kolisch [Kol96b]	42.84	41.84	40.63
sampling - random	Kolisch [Kol95]	44.46	43.05	41.44
sampling - random	Kolisch [Kol95]	49.25	47.61	45.60

TAB. 2.9 – Ecart moyen au makespan trivial pour les instances de la PSPLIB de type j120 (tableau original extrait de [KH06])

Le tableau 2.9, qui présente le classement des méthodes sur les instances à 120 jobs, fait apparaître une méthode qui n'apparaît pas dans les deux autres tableaux : celle de [MMS02] qui se classe en neuvième position. Au contraire [TL00] ne fournit pas de résultats pour ces instances. Le GRASP est légèrement moins bien classé que dans les tableaux précédents mais reste dans la première moitié du classement. Ce classement est satisfaisant pour une méthode qui n'est pas dédiée directement à la résolution du RCPSP.

2.2.5 Conclusion

Dans cette section, nous avons établi le lien entre un problème d'ordonnancement sous contraintes de ressources (RCPSP) et un problème de flots en montrant comment une solution peut être représentée comme un multiflot. Cette modélisation nous permet de concevoir des heuristiques de construction et des recherches locales qui tirent profit de la représentation à l'aide des flots. Notre méthode a été testée sur les instances de la PSPLIB et les résultats montrent son efficacité. Une étude statistique des instances à 30 jobs de la PSPLIB a également été réalisée pour mettre en évidence les caractéristiques des instances. Il apparaît que, pour de nombreuses instances, la borne inférieure triviale est égale au makespan optimal.

L'intérêt de cette modélisation est qu'elle peut être étendue à des problèmes d'ordonnancement sous contraintes de ressources qui possèdent des contraintes supplémentaires. La section suivante montre comment les algorithmes décrits ici peuvent êtres adaptés au RCPSP avec time lags conditionnels.

2.3 RCPSP avec *time lags* conditionnels

Cette section est dédiée à une extension du RCPSP : le RCPSP avec *time lags* conditionnels (RCPSPTL). Dans ce problème, on a besoin de connaître explicitement la manière dont sont échangées les ressources entre les activités. Le modèle de flots décrit précédemment est donc tout à fait approprié pour la modélisation et la résolution de ce problème.

2.3.1 RCPSP avec *time lags*

A notre connaissance, le RCPSP avec *time lags* conditionnels n'a jamais été étudié. Néanmoins, il existe des extensions du RCPSP avec *time lags* non conditionnels (minimaux ou maximaux). Nous présentons ici les contraintes liées aux *time lags* non conditionnels, avant d'introduire notre extension particulière.

2.3.1.1 *Time lags* minimaux

Dans le RCPSP classique, une activité doit finir avant que ses successeurs ne commencent. Ce concept basique de précédence peut être étendu par des *time lags* minimaux $lag_{min}(i,j)$ entre la date de début S_j de l'activité j et la date de fin F_i de l'activité i. On a alors la relation $S_j \geq F_i + lag_{min}(i,j)$: j doit commencer au moins $lag_{min}(i,j)$ unités de temps après la fin de i. Ces time lags peuvent, par exemple, représenter des temps d'attente dus à des contraintes physiques, des contraintes de maintenance ...

Ces contraintes ont été considérées, entre autres, par [CS05], [Kle00], [KS00], [KS99a], [Kol00] et [Van06].

2.3.1.2 *Time lags* maximaux

Des *time lags* maximaux $lag_{max}(i,j)$ peuvent également être considérés entre la date de fin F_i de l'activité i et la date de début S_j de l'activité j. On a alors la relation $S_j \leq F_i + lag_{max}(i,j)$: j ne doit pas commencer plus de $lag_{max}(i,j)$ unités de temps

après la fin de i. Le RCPSP avec *time lags* minimaux et maximaux est généralement noté RCPSP/max. Notons que la présence de *time lags* maximaux peut rendre une instance infaisable.

Parmi les auteurs qui ont étudié le RCPSP/max, on peut citer [DPH00], [BMR88], [COS02], [NSZ06] et [ST00].

2.3.2 Définition du RCPSP avec *time lags* conditionnels

Une instance du RCPSPTL contient les mêmes composants qu'une instance du RCPSP classique (ensemble d'activités A, ensemble de ressources R et contraintes de précédence $\ll$), l'originalité vient du fait qu'on ajoute des *time lags conditionnels* qui ont la signification suivante :

- On suppose que les activités sont traitées sur des lieux différents et les ressources (supposées renouvelables) doivent donc circuler entre ces lieux, ce qui engendre un coût (durée) de transport. On suppose que toutes ces ressources sont entreposées dans un dépôt. On doit les transporter vers les lieux de traitement au début du projet puis les ramener au dépôt à la fin du projet. Le projet est terminé quand toutes les activités sont terminées et que les ressources sont revenues au dépôt ;

- La durée du transport est modélisée par un *time lag* conditionnel : si i transmet une quantité non nulle de ressources vers j ou si $i \ll j$ (ce qui signifie en général que le résultat de l'activité i est transmis à l'activité j), alors le transport de cette ressource impose un délai noté $lag(i,j)$, entre la fin de i et le début de j. Le transport initial du dépôt vers les lieux de traitement est modélisé par un *time lag* entre la source et les activités tandis que le transport final, pour le retour des ressources au dépôt, est modélisé par un *time lag* entre les activités et le puits.

Le problème consiste alors à calculer la date de début Δ_i de chaque activité $i \in A$ et les transferts de ressources tels que :

- les contraintes de précédence sont respectées :

$$\forall i \in A^*,\ \forall j \in A^*,\ i \ll j \Rightarrow \Delta_i + d_i + lag(i,j) \leq \Delta_j;$$

- la somme des ressources utilisées par les activités en traitement et des ressources en cours de transfert ne dépasse pas la quantité maximale de ressources disponibles :

$$\forall k \in R,\ \forall t \in [0, C_{max}],\ \sum_{\substack{i \in A^* \\ \Delta_i \leq t < \Delta_i + d_i}} r_{ik} + \sum_{(i,j) \in Lag_t} F_{k(i,j)} \leq M_k;$$

  où Lag_t est l'ensemble des couples d'activités (i,j) tel que i transfère de la ressource à j à l'instant t.

- le makespan $C_{max} = \max\limits_{\substack{i \in A \\ j \in succ(i)}} \big(\Delta_i + d_i + lag(i,j)\big)$ est minimal.

Remarque 2

Contrairement aux définitions classiques de RCPSP avec time lag que l'on rencontre dans la littérature, la particularité de notre modèle réside dans le fait que les time lags sont conditionnels : ils sont à prendre en compte uniquement dans le cas où il y a transfert de ressources entre les activités.

Remarque 3
Les time lags conditionnels sont une approche possible pour la modélisation du transport de ressources entre des lieux de traitements différents. On suppose dans notre modèle qu'un time lag entre deux activités ne dépend pas du type de ressources transférées ni de la contrainte liant ces activités (contrainte de flot ou contrainte de précédence). Néanmoins, le modèle et les méthodes que nous proposons peuvent s'adapter facilement pour prendre en compte ces contraintes supplémentaires.

2.3.3 Reformulation du RCPSPTL en un problème de multiflot

De même que pour le RCPSP classique, une solution du RCPSPTL peut être définie par un graphe $G = (X, E)$ et un multiflot F tels que :

- l'ensemble des sommets X est l'ensemble A^* ;
- l'ensemble des arcs E est l'ensemble des arcs induits par une contrainte de précédence et des arcs induits par un transfert de ressources : $E = E_{\ll} \cup E_F$.

Les arcs (i, j) de E sont valués par la somme $\big(d_i + lag(i, j)\big)$ de la durée de l'activité i et du *time lag* entre l'activité i et l'activité j. Une solution du RCPSPTL est donc entièrement définie par le graphe G et le multiflot F : les dates de début des activités s'obtiennent par un calcul des plus longs chemins dans G et les transferts de ressources sont donnés par F.

Remarque 4
La seule différence entre le graphe défini pour le RCPSP classique et le RCPSPTL réside dans la valuation des arcs : dans le RCPSP classique un arc $(i, j) \in E$ est valué par d_i alors que dans le RCPSPTL un arc $(i, j) \in E$ est valué par $d_i + lag(i, j)$. La modélisation du problème change donc très peu. Néanmoins les algorithmes du problème classique ne peuvent pas être adaptés directement (en changeant simplement la valuation des arcs) puisque dans le problème avec time lags conditionnels, la valuation d'un arc dépend non plus uniquement de son sommet origine mais également du sommet destination.

Remarque 5
Il est clair qu'une instance du RCPSP classique peut être transformée en une instance du RCPSPTL dans laquelle tous les time lags sont nuls.

2.3.4 Algorithme d'insertion : Génération d'une solution initiale

La formulation du RCPSPTL étant très proche de la formulation du problème classique, la même démarche est utilisée pour la résolution. Les algorithmes ont été adaptés pour prendre en compte les *time lags* conditionnels.

2.3.4.1 Nouvelles formules de récurrence pour le calcul de π et de $\bar{\pi}$

- $\pi(i)$ la date de fin au plus tôt de i, $\pi(i)$ se définit récursivement par :

$$\begin{cases} \pi(s) = 0 \\ \forall i \in A^* - \{s\},\ \pi(i) = \max\limits_{j \in pred(i)} (\pi(j) + d_i + lag(j,i)) \end{cases}$$

- $\bar{\pi}(i)$ la longueur du plus long chemin entre i et p, $\bar{\pi}(i)$ se définit récursivement par :

$$\begin{cases} \bar{\pi}(p) = 0 \\ \forall i \in A^* - \{p\},\ \bar{\pi}(i) = \max\limits_{j \in succ(i)} (\bar{\pi}(j) + d_i + lag(i,j)) \end{cases}$$

2.3.4.2 Insertion optimale

On considère le cas $|R| = 1$ (un seul flot).

Soit x_0 l'action à insérer de durée d_{x_0} et consommation ρ. Soit une coupe (U, V) telle que $\sum\limits_{u \in U} out(u) = \sum\limits_{v \in V} in(v) \geq \rho$. Un flot h est dit compatible avec l'insertion de x_0 dans (U, V) s'il vérifie la propriété (P_{insert}) suivante (même propriété que pour le RCPSP classique) :

$$(P_{insert}) \begin{cases} \forall u \in U,\ out(u) = \sum\limits_{v \in V \cup \{x_0\}} h_{(u,v)} \\ \forall v \in V,\ in(v) = \sum\limits_{u \in U \cup \{x_0\}} h_{(u,v)} \\ \rho = \sum\limits_{v \in V} h_{(x_0,v)} = \sum\limits_{u \in U} h_{(u,x_0)} \end{cases}$$

On rappelle que $H = \{u \in U, v \in V / u \ll v \text{ ou } h_{(u,v)} \neq 0\}$ et $H_2 = \{u \in U, v \in V / (u \ll x_0 \text{ ou } h_{(u,x_0)} \neq 0) et (x_0 \ll v \text{ ou } h_{(x_0,v)} \neq 0)\}$.

Pour un flot h qui vérifie P_{insert} on pose :

$C_1(h) = \max\limits_{\substack{u \in U \times H \\ v \in V \times H}} \big(\pi(u) + \bar{\pi}(v) + lag(u,v)\big)$;

$C_2(h) = \max\limits_{\substack{u \in U \times H_2 \\ v \in V \times H_2}} \big(\pi(u) + \bar{\pi}(v) + d_{x_0} + lag(u,x_0) + lag(x_0,v)\big)$;

$C'_{max}(h) = \max\big(C_1(h), C_2(h)\big)$.

Définition du problème

Problème de l'insertion : *Etant donné un graphe biparti (X, E) tel que $X = U \cup V$ et une activité x_0 qui n'est pas dans X, trouver un flot compatible avec l'insertion de x_0 dans (U, V) qui minimise $C'_{max}(h)$.*

Solution du problème

Contrairement au RCPSP classique, on ne donne pas ici de méthode pour insérer directement x_0 et donc fixer les valeurs de flot. Le problème est résolu en deux étapes :

étape 1 on calcule un flot h compatible avec l'insertion de x_0 dans (U, V) et on insère x_0 (sans tenir compte de la valeur $C'_{max}(h)$) ;

étape 2 on cherche un chemin γ qui permet de rediriger le flot afin d'améliorer $C'_{max}(h)$.

On note ρ la consommation en ressource de x_0. On appelle valeur de raccrochage pour le flot h la valeur π_r telle que :

$$\pi_r = \max_{\substack{u \in U / h_{(u,x_0)} > 0 \\ ou\ (u,x_0) \in E_{\ll}}} \big(\pi(u) + lag(u, x_0)\big)$$

Il est clair que π_r doit être tel que $\sum_{\substack{u' \in U / \\ \pi(u) + lag(u,x_0) \leq \pi_r}} out(u') \geq \rho$ pour que x_0 reçoive suffisamment de flot.

Ayant défini un flot h compatible avec l'insertion de x_0, on pose $\pi(x_0) = \pi_r + d_{x_0}$. On cherche alors un chemin *de redirection* qui permet de modifier les flots dans la coupe de sorte que la valeur π_r n'augmente pas et la valeur $C'_{max}(h)$ diminue.

On suppose que l'arc qui donne la valeur de $C'_{max}(h)$ est (u_0, v_0), c'est donc l'arc qu'on tente de supprimer en redirigeant les flots. Un chemin de redirection pour le flot h et la valeur π_r est alors défini comme une séquence $\gamma = \{u_0, v_0, u_1, v_1 \dots u_n = u_0\}$ telle que :

(1) $\forall i \in [0 \dots n],\ u_i \in U \cup \{x_0\}$;

(2) $v_0 \in V$ et $\forall i \in [1 \dots n],\ v_i \in V \cup \{x_0\}$;

(3) $\forall i \in [0 \dots n-1],\ h_{u_i, v_i} > 0$;

(4) x_0 est au plus présent 2 fois dans γ et s'il est présent 2 fois, il apparaît une fois comme u_i, une fois comme v_i et de manière non consécutive ;

(5) les activités de $U \cup V$ sont au plus présentes une fois dans γ ;

(6) $\pi_{t_r}(u_0) + \bar{\pi}(v_0) + lag(u_0, v_0) = \max_{\substack{i \in [0 \dots n-1] / v_i \in V \\ et\ (u_i, v_i) \notin E}} (\pi_{t_r}(u_i) + \bar{\pi}(v_i) + lag(u_i, v_i))$;

(7) $\forall i \in [0 \dots n-1]$, tel que $v_i \in V$ et $(u_{i+1}, v_i) \notin E$, on a $\pi(u_{i+1}) + \bar{\pi}(v_i) + lag(u_{i+1}, v_i) < \pi(u_0) + \bar{\pi}(v_0) + lag(u_0, v_0)$;

(8) $\forall i \in [0 \dots n-1]$ tel que $v_i = x_0$ on a $\pi(u_{i+1}) + lag(u_{i+1}, v_i) \leq \pi_r$.

Les points (1) et (2) donnent les ensembles d'appartenance des sommets du chemin. Le point (3) stipule que les arcs de la forme (u_i, v_i) doivent porter du flot. Les points (4) et (5) donnent le nombre d'occurrence maximale des sommets de la coupe et de x_0 dans γ. Dans un souci de simplification, on dit qu'un arc (u, v) est plus coûteux qu'un arc (u', v') si $\pi(u) + \bar{\pi}(v) + lag(u, v) > \pi(u') + \bar{\pi}(v') + lag(u', v')$, alors (6) stipule que parmi les arcs (u_i, v_i), le plus coûteux est (u_0, v_0). (7) stipule que les arcs de la forme (u_{i+1}, v_i) sont moins coûteux que l'arc (u_0, v_0). Enfin (8) rappelle la signification de π_r.

Etant donné un tel chemin de redirection γ, le processus de redirection consiste à ajouter le maximum de flots sur les arcs (u_{i+1}, v_i) et à en retirer le maximum sur les arcs (u_i, v_i) tel que décrit par l'algorithme 23 et illustré par l'exemple 8.

Algorithme 23 : RedirectionTL

Entrées : $\gamma = (u_0, v_0, u_1, v_1 \ldots u_n = u_0)$, h
Sorties : h

1. $\delta = \min\limits_{i \in [0 \ldots n-1]} h_{(u_i,v_i)}$;
2. **Pour** $i = 0$ *à* $n-1$ **faire**
3. $\quad h_{(u_i,v_i)} \leftarrow h_{(u_i,v_i)} - \delta$;
4. $\quad h_{(u_{i+1},v_i)} \leftarrow h_{(u_{i+1},v_i)} + \delta$;

Notons que si l'arc qui donne la valeur de $C'_{max}(h)$ est imposée par des contraintes de précédence $\ll$ sa valeur ne peut pas diminuer, il est alors inutile de chercher un chemin de redirection.

Exemple 8. *Redirection des flots*

On suppose que x_0 (de durée $d_{x_0} = 2$) a été inséré dans une coupe (U, V). Le flot h et les valeurs qui interviennent dans le calcul de $C'_{max}(h)$ sont donnés par la figure 2.14(a).

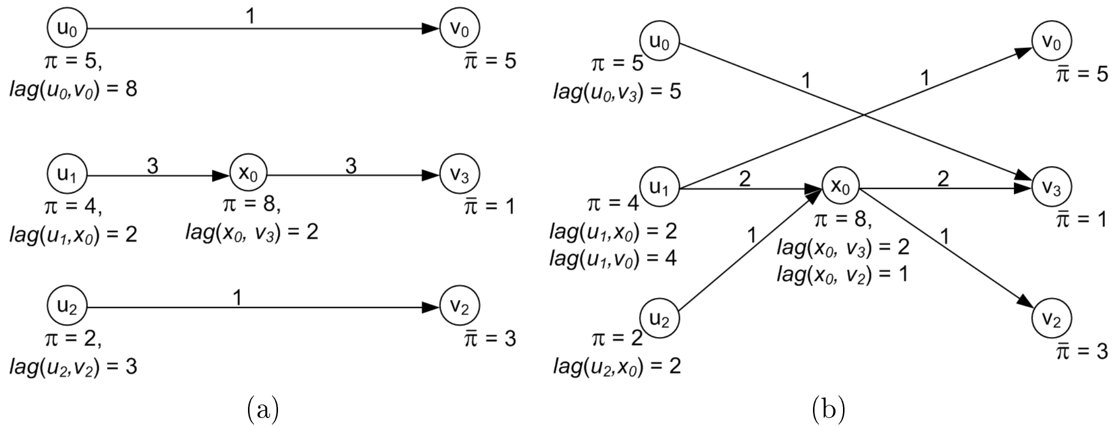


FIG. 2.14 – Redirection du flot associé à l'exemple 8

Sur l'exemple de la figure 2.14 on a :
$C'_{max}(h) = \pi(u_0) + \bar{\pi}(v_0) + lag(u_0, v_0) = 18$;
$\pi_r = \pi(u_1) + lag(u_1, x_0) = 6$;
$\pi(x_0) = \pi_r + d_{x_0} = 8$.

On souhaite donc supprimer l'arc (u_0, v_0). Un chemin de redirection est donné par $\gamma = (u_0, v_0, u_1, x_0, u_2, v_2, x_0, v_3, u_0)$, dans lequel x_0 joue le rôle de v_1 et de u_3. On applique l'algorithme 23 : $\delta = 1$, ainsi on enlève une unité de flot sur les arcs (u_0, v_0), (u_1, x_0), (u_2, v_2), (x_0, v_3) et on ajoute une unité de flot sur les arcs (u_1, v_0), (u_2, x_0), (x_0, v_2), (x_0, v_3). On obtient alors le flot h représenté figure 2.14(b). On a réussi à supprimer l'arc (u_0, v_0) et le nouveau $C'_{max}(h)$ est égal à $\pi(u_1) + \bar{\pi}(v_0) + lag(u_1, v_0) = 13$.

Algorithme 24 : InsertionTL

Entrées : U, V, x_0, h
Sorties : h_{best}

1 $C_{best} \leftarrow +\infty$;
2 **Calculer** les valeurs de raccrochage π_r possibles pour l'insertion de x_0 ;
3 **pour chaque** π_r **faire**
4 $\quad h' \leftarrow h$;
5 $\quad$ **Calculer** un flot h' initial dans la coupe (U, V) dont la valeur de raccrochage est π_r ;
6 $\quad$ *stop* $\leftarrow$ *faux* ;
7 $\quad$ **Tant que** $stop = faux$ **faire**
8 $\quad\quad$ Soit $(u_0, v_0) \in U \cup \{x_0\} \times V$ l'arc tel que :
9 $\quad\quad\quad C'_{max}(h') = \pi(u_0) + \bar{\pi}(v_0) + lag(u_0, v_0)$;
10 $\quad\quad$ **Chercher** un chemin de redirection $\gamma = (u_0, v_0, \dots u_n = u_0)$ pour h' et π_r ;
11 $\quad\quad$ **Si** γ *n'existe pas* **alors**
12 $\quad\quad\quad$ stop = vrai ;
13 $\quad\quad$ **sinon**
14 $\quad\quad\quad h'$ = **RedirectionTL**(γ, h') ;
15 $\quad$ **Si** $C'_{max}(h') < C_{best}$ **alors**
16 $\quad\quad C_{best} \leftarrow C'_{max}(h')$;
17 $\quad\quad h_{best} \leftarrow h'$;

Théorème 4
L'algorithme 24 est optimal pour le problème de l'insertion.

Démonstration

Soit un flot h optimal pour le problème de l'insertion et π_r sa valeur de raccrochage. Soit h' un flot calculé par l'algorithme 24 uniquement pour la valeur de raccrochage π_r. Si $C'_{max}(h') > C'_{max}(h)$ la différence h - h' est un flot qui fait apparaître une ou plusieurs décompositions en flot-cycles qui induisent des chemins de redirection. Or l'algorithme 24 utilise tous les chemins de redirection possibles pour améliorer h' donc h - $h' = 0$. □

2.3.4.3 Insertion rapide

De même que pour le RCPSP classique, la procédure d'insertion est appelée pour chaque action x_0 à insérer et pour chaque coupe compatible avec l'insertion de x_0. Elle doit donc être assez rapide pour que le temps d'exécution de l'algorithme ne soit pas trop élevé. La recherche d'un chemin est une opération coûteuse en temps de calcul. On peut remplacer cette étape par un procédé heuristique plus rapide mais qui ne garantit pas l'optimalité. On décrit ici ce procédé qui s'avère être efficace dans les expérimentations numériques.

Pour tout $u \in U$, on note $res(u)$ la plus petite valeur telle que l'inégalité suivante est vérifiée :

$$\sum_{\substack{v \in V / \\ \bar{\pi}(v) + lag(u,v) \leq res(u)}} in(v) \geq out(u)$$

La valeur $res(u)$ correspond à la contribution minimale de u à $C'_{max}(h)$, c'est-à-dire

qu'on se place dans le cas où on crée, à cause du flot h, des arcs (u, v) dont le coût $\pi(u) + \bar{\pi}(v) + lag(u, v)$ est le plus faible possible.

La procédure d'insertion rapide raccroche tout d'abord x_0 à U en choisissant les activités $u \in U$ dont la valeur $res(u)$ est la plus grande possible. Ensuite la procédure continue de même en choisissant à chaque itération une activité $u \in U \cup \{x_0\}$, telle que $out(u) > 0$ et $res(u)$ est la plus grande possible. On met du flot entre l'activité u et l'activité $v \in V$ de plus faible $\bar{\pi}$. Le schéma algorithmique est le même que pour l'algorithme 24, seule la boucle principale, qui traite l'insertion, est modifiée (voir algorithme 25).

Algorithme 25 : InsertionRapideTL (boucle principale)

Entrées : (U, V), x_0 de consommation ρ et durée d_{x_0}, π_r
Sorties : h

1 //Raccrochage de x_0 à U :
2 $r \leftarrow \rho$;
3 **Tant que** $r \neq 0$ **faire**
4   **Calculer** $u \in U$ telle que $\pi(u) + lag(u, x_0) \leq \pi_r$ et $\pi(u) + res(u)$ est maximale ;
5     $\delta \leftarrow \min(out(u), r)$;
6     $h_{u,x_0} \leftarrow \delta$;
7     $out(u) \leftarrow out(u) - \delta$;
8     $r \leftarrow r - \delta$;
9 $out(x_0) \leftarrow \rho$;
10 $\pi(x_0) \leftarrow \pi_r + d_{x_0}$;
11 //calcul des flots dans la coupe $(U \cup \{x_0\}, V)$:
12 **pour chaque** $u \in (U \cup \{x_0\}, V)$ **faire**
13   **Calculer** $u \in U$ telle que $out(u) > 0$ et $\pi(u) + res(u)$ est maximale ;
14   **Tant que** $out(u) \neq 0$ **faire**
15   **Calculer** $v \in V$ tel que $in(v) \neq 0$ et $\bar{\pi}(v) + lag(u, v)$ est minimal ;
16         $\delta \leftarrow \min(out(u), in(y))$;
17         $h_{u,v} \leftarrow \delta$;
18         $out(u) \leftarrow out(u) - \delta$;
19         $in(v) \leftarrow in(v) - \delta$;

2.3.4.4 Insertion multiflot

Dans le cas du RCPSPTL multiflot, on utilise les mêmes techniques pour passer d'un flot unique à un multiflot que dans le cas du RCPSP : l'algorithme d'insertion `InsertionTL` (ou `InsertionRapideTL`) est simplement appliqué pour chaque type de ressources et les valeurs de raccrochage sont calculées pour l'ensemble des ressources (voir algorithme 26).

De même que dans le cas du RCPSP classique, il est intéressant de faire porter du flot aux arcs de $E_{\ll}$ ainsi qu'aux arcs (u, v) de plus petit $\pi(u) + \bar{\pi}(v) + lag(u, v)$.

Algorithme 26 : InsertionMultiTL

Entrées : U, V, x_0, $F = (f_1, \ldots, f_m)$
Sorties : F_{best}

1 $C_{best} \leftarrow +\infty$;
2 Calculer l'ensemble Π_r des valeurs de raccrochage possibles pour x_0 dans U ;
3 **pour chaque** $\pi_r \in \Pi_r$ **faire**
4 $\quad F' \leftarrow F : \forall k \in R, f'_k \leftarrow f_k$;
5 $\quad$ **pour chaque** *type de ressource* $k \in R$ **faire**
6 $\quad\quad$ InsertionTL(U, V, x_0, f'_k) ;
7 $\quad$ **Si** $C'_{max}(F') < C_{best}$ **alors**
8 $\quad\quad F_{best} \leftarrow F'$;
9 $\quad\quad C_{best} \leftarrow C'_{max}(F')$;

2.3.4.5 Algorithme d'évaluation

L'algorithme d'évaluation est structuré de la manière que dans le cas du RCPSP classique (section 2.2.2.7), seule change la technique d'insertion de l'élément x_0 : la fonction `InsertionMulti` est donc remplacée par `InsertionMultiTL`.

2.3.5 Recherche locale

De même que pour le RCPSP classique, les procédures décrites précédemment peuvent également être utilisées comme opérateur de recherche locale. Les algorithmes sont les mêmes que pour le RCPSP classique (section 2.2.3), seul l'algorithme de suppression est modifié (voir algorithme 27).

Algorithme 27 : SuppressionTL

Entrées : x_0, F
Sorties : F

1 **Construire** la coupe (U, V) : U est l'ensemble des activités qui donnent du flot à x_0 et V est l'ensemble des activités qui prennent du flot de x_0 ;
2 **Supprimer** dans F les flots relatifs à la coupe ;
3 Soit y_0 une activité fictive de durée nulle et consommation nulle ;
4 $F \leftarrow$ **InsertionMultiTL**(U, V, y_0, F) ;

2.3.6 Résultats numériques

On présente les résultats des tests pour l'algorithme d'évaluation du RCPSPTL. Les tests ont été effectués sur un PC AMD Opteron, 2.1GHz. On a utilisé deux jeux d'instances. Le premier est constitué d'instances de la PSPLIB. Le second est construit de manière à connaître la solution optimale pour chaque instance et ainsi quantifier l'efficacité de notre algorithme. La procédure d'évaluation, pour laquelle nous présentons des résultats, utilise pour l'insertion la procédure *InsertionTL* de la manière suivante : le flot est initialisé grâce à la procédure d'insertion rapide, puis amélioré grâce à la recherche de chemins de

redirection. On se contente de chercher des chemins de longueur 4, ce qui est plus rapide que la recherche de chemins de longueur quelconque et donne des résultats de qualité similaire.

2.3.6.1 Instances modifiées de la PSPLIB

On utilise les instances de la PSPLIB conçues pour le RCPSP avec *time lags* minimaux et maximaux. Les *time lags* maximaux sont supprimés et les *time lags* minimaux sont conservés de manière à obtenir une instance pour notre problème. A notre connaissance, il n'existe pas de benchmark pour le RCPSP avec *time lags* conditionnels. Nous comparons nos résultats avec ceux des instances classiques même s'il est clair que la conditionnalité des *time lags* change radicalement le problème. Cependant, le fait que les *time lags* soient conditionnels doit conduire à une solution de plus faible makespan que dans le cas non conditionnel. On s'assure que c'est le cas et on compare les différences.

Les groupes d'instances choisis sont UBO10, UBO20, UBO50, UBO100 qui comptent respectivement 10, 20, 50 et 100 activités. Chaque groupe contient 90 instances, le tableau 2.10 donne les résultats moyens par groupe d'instances. Dans le cas des instances avec *time lags* maximaux, il peut arriver qu'il n'existe pas de solution, dans ce cas l'instance n'intervient pas dans la moyenne. La colonne *gap_ opt* donne l'écart moyen en pourcentage entre la valeur de la solution trouvée par notre méthode et la valeur de la solution optimale. La colonne *gap_LB* donne l'écart moyen en pourcentage entre la valeur de la solution trouvée par notre méthode et la valeur de la borne inférieure fournie par la PSPLIB.

D'après les résultats on constate, qu'effectivement, l'ordonnancement des projets est fortement modifié si on considère des *time lags* conditionnels plutôt que des *time lags* classiques.

groupe	Δ	gap_opt (%)	gap_LB (%)	cpu total (s)
10	100	-27.71	-26.51	0.04
20	100	-41.09	-37.37	0.22
50	100	-55.20	-40.27	3.53
100	100	-57.59	-40.74	48.93
10	500	-27.75	-26.55	0.20
20	500	-41.61	-37.98	1.11
50	500	-55.74	-40.81	17.64
100	500	-58.05	-41.13	245.28

TAB. 2.10 – Résultats moyens sur 5 groupes d'instances de 10 à 100 activités pour Δ réplications de la procédure `Evaluer`

2.3.6.2 Instances générées

Les instances que nous avons construites impliquent un seul type de ressources. Ces instances sont construites de la manière suivante :

Etant donnés la quantité de ressources disponibles et le makespan optimal qu'on souhaite atteindre, on génère un ordonnancement $\mathcal{O}$ « parfait » pour le RCPSP classique : à chaque instant toutes les ressources sont utilisées. Les activités sont générées itérativement avec une durée et une consommation aléatoires mais de manière à optimiser l'utilisation des ressources. On connait alors, les dates de début optimales pour les activités. Un exemple

de diagramme de Gantt pour un tel ordonnancement est donné figure 2.15, la quantité maximale de ressources disponibles est 10 et le makespan vaut 20.

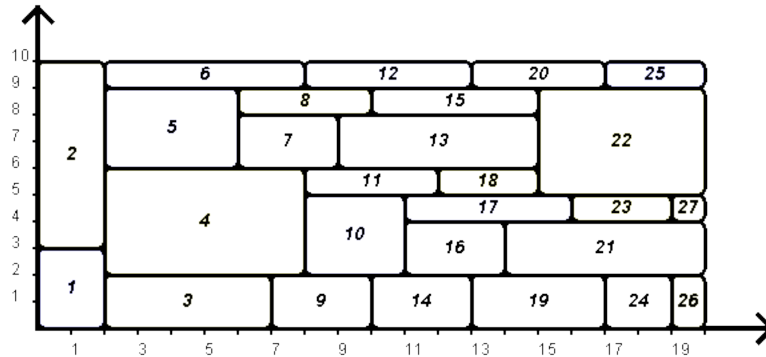


FIG. 2.15 – Exemple de solution « parfaite » pour le RCPSP classique avec une ressource disponible en quantité 10

Ensuite, des contraintes de précédence sont ajoutées aléatoirement mais de manière à garder l'ordonnancement $\mathcal{O}$ valide : on ne met une contrainte de précédence de type $i \ll j$ entre les activités i et j que si j commence après la fin de i dans $\mathcal{O}$. Enfin des *time lags* sont ajoutés de la manière suivante : si deux activités i et j sont consécutives dans l'ordonnancement $\mathcal{O}$ alors $lag(i, j) = 0$, sinon $lag(i, j)$ est généré aléatoirement par une loi uniforme entre une valeur minimale et une valeur maximale données. Si deux activités i et j sont telles que $i \ll j$, alors on s'assure que $lag(i, j)$ est inférieur à la différence entre la date de début de j et la date de fin de i de manière à ne pas décaler la date de début de j dans $\mathcal{O}$. De cette manière, on obtient une instance pour le RCPSPTL dont on connaît la valeur du makespan optimal.

Pour réaliser les tests, on a généré 10 ordonnancements « parfaits » pour le RCPSP avec 23.6 activités en moyenne (de 15 à 29). Pour un même ordonnancement, on génère six instances qui diffèrent, soit par le nombre moyen de contraintes de précédence, soit par la valeur moyenne des *time lags*. De cette manière, on peut observer l'impact du nombre de contraintes de précédence et de la valeur des *time lags* sur l'efficacité de l'algorithme. Les résultats moyens par groupe sont présentés dans les tableaux 2.11 (100 réplications de l'algorithme) et 2.12 (500 réplications). Dans ces tableaux, les colonnes ont la signification suivante :

- groupe : numéro du groupe de 10 instances ;
- successeurs : nombre de successeurs imposés pour une activité ;
- TL : valeur du *time lag* imposé entre deux activité non consécutives ;
- min : meilleur makespan trouvé parmi les N réplications ;
- max : pire makespan trouvé parmi les N réplications ;
- moyenne : makespan moyen trouvé pour les N réplications ;
- gap : pourcentage d'erreur entre la meilleure solution trouvée et la solution optimale ;
- cpu total : temps total en secondes pour les N réplications ;

groupe	successeurs	TL	min	max	moyenne	gap (%)	cpu total (s)
1	entre 0 et 2	entre 0 et 2	22.60	33.40	26.58	13.00	0.11
2	entre 0 et 5	entre 0 et 2	22.30	33.00	26.04	11.50	0.11
3	entre 0 et 2	entre 3 et 5	20.00	40.40	26.18	0.00	0.13
4	entre 0 et 5	entre 3 et 5	20.00	37.30	25.28	0.00	0.13
5	entre 0 et 2	entre 6 et 9	20.00	35.30	21.69	0.00	0.13
6	entre 0 et 5	entre 6 et 9	20.00	32.50	21.42	0.00	0.13

Tab. 2.11 – 100 réplications

groupe	successeurs	TL	min	max	moyenne	gap (%)	cpu total (s)
1	entre 0 et 2	entre 0 et 2	22.00	36.40	26.69	10.00	0.57
2	entre 0 et 5	entre 0 et 2	21.60	34.00	25.96	8.00	0.57
3	entre 0 et 2	entre 3 et 5	20.00	41.60	26.08	0.00	0.68
4	entre 0 et 5	entre 3 et 5	20.00	40.10	25.27	0.00	0.66
5	entre 0 et 2	entre 6 et 9	20.00	38.30	21.74	0.00	0.67
6	entre 0 et 5	entre 6 et 9	20.00	36.80	21.59	0.00	0.65

Tab. 2.12 – 500 réplications

On constate, d'une part, que les temps de calcul restent très courts malgré l'ajout des *time lags*. D'autre part, plus les *time lags* sont longs (ils prennent donc plus de poids que la durée des activités) plus l'algorithme semble efficace. Ils orientent donc l'algorithme dans le choix de l'insertion a réaliser.

2.3.7 Conclusion

Nous nous sommes intéressé à une extension du RCPSP dans laquelle des contraintes temporelles supplémentaires sont à prendre en compte. Ces contraintes modélisent un temps de transport obligatoire pour transférer les ressources entre les activités. Elles dépendent donc des activités concernées. A notre connaissance, cette extension n'a jamais été étudiée. Nous proposons, pour résoudre ce problème, une approche basée sur un multiflot. Les multiflots sont particulièrement bien adaptés à la modélisation des échanges entre activités. Nous avons adapté les procédures que nous avions élaborées pour le RCPSP. Cela a permis de montrer que notre méthode s'adapte facilement à des extensions, en particulier des extensions qui modélisent des échanges de ressources. De ce fait, une autre extension à été envisagée : l'ajout de contraintes financières. Les contraintes financières peuvent s'exprimer comme des flux financiers entre les ressources. Il est donc possible d'utiliser des flots pour les représenter.

2.4 Perspectives : Ajout d'une ressource non renouvelable

Jusqu'à présent, nous nous sommes intéressé à l'ordonnancement de projet sous contraintes de ressources renouvelables. Nous envisageons maintenant de prendre en compte une ressource supplémentaire non renouvelable : une ressource financière. Dans ce problème, une activité a besoin d'une certaine quantité d'argent pour être exécutée. La quantité d'argent dépensée pour traiter cette activité est perdue au moment où débute l'activité.

Néanmoins, son exécution entraîne un revenu qui est perçu après la fin du traitement de l'activité (avec un délai éventuel). Cette ressource financière peut être gérée grâce à des processus d'emprunts.

Une idée pour traiter ce problème est de représenter explicitement les échanges de ressources renouvelables et non renouvelables entre les activités. Pour cela on peut étendre le modèle de flots que nous avons proposé pour le RCPSP.

2.4.1 Notations

Nous reprenons les notations de la section 2.2 et nous ajoutons les notations concernant la ressource financière. Toutes les activités i ont besoin de Φ_i unités d'argent et en produisent ψ_i. Il y a un délais $\delta(i)$ entre la fin de i et le moment où ψ_i est effectivement produit et disponible. On dispose d'un capital initial ψ_s. On note $\phi_{(i,j)}$ la quantité de ressource financière transmise de i vers j.

Les contraintes suivantes sont alors ajoutées à la propriété (P_{flot}) :

$$\begin{cases} \forall i \in A \cup \{s\}, \sum_{j \in A \cup \{p\}} \phi_{(i,j)} = \psi_i \\ \forall j \in A, \sum_{i \in A \cup \{s\}} \phi_{(i,j)} = \Phi_j \end{cases}$$

2.4.2 Adaptation de la technique d'insertion

Les techniques d'insertion pour les ressources renouvelables sont inchangées. Nous envisageons d'adapter ces techniques pour la ressource financière. Les différents problèmes évoqués dans la section 2.2 sont donc légèrement modifiés.

2.4.2.1 Problème de la coupe

Nous envisageons de calculer le multiflot dans une coupe (U, V) en deux temps :
- tout d'abord, on calcule le multiflot concernant les ressources renouvelables avec la technique proposée en section 2.2 ;
- ensuite, on calcule les flux financiers avec la même technique à la différence que ϕ doit respecter la propriété $P_{\phi cross}$ au lieu de la propriété P_{cross} utilisée pour les ressources renouvelables. La propriété $P_{\phi cross}$ est définie par :

$$(P_{\phi cross}) \begin{cases} \text{Il n'existe pas de sommets } u,\ u' \text{ dans } U \text{ et } v, v' \text{ dans } V \text{ tels que :} \\ \quad \phi_{(u,v)} > 0 \\ \quad \phi_{(u',v')} > 0 \\ \quad \pi(u') + \delta(u') > \pi(u) + \delta(u) \text{ et } \bar{\pi}(v') > \bar{\pi}(v) \end{cases}$$

2.4.2.2 Problème de l'insertion

Soit une coupe (U, V) et une activité x_0 à insérer dans la coupe. L'activité de raccrochage u_r est choisie de manière à prendre en compte les contraintes financières. On définie

donc une nouvelle propriété ($P_{\phi raccro}$) pour le flux ϕ :

$$(P_{\phi raccro}) \begin{cases} \forall u \in U, u' \in U, \\ \big(\pi(u') + \delta(u') < \pi(u) + \delta(u) \leq \pi(u_r) + \delta(u_r) \text{ et } \phi_{(u',x_0)} \neq 0\big) \Rightarrow \phi_{(u,x_0)} = out(u) \\ \forall u \in U,\ \pi(u) + \delta(u) > \pi(u_r) + \delta(u_r) \Rightarrow \phi_{(u,x_0)} = 0 \end{cases}$$

Ainsi l'activité à insérer est raccrochée à u_r de sorte que le multiflot associé aux ressources renouvelables vérifie (P_{raccro}) et le flux associé à la ressource financière vérifie ($P_{\phi raccro}$).

2.4.3 Conclusion

Nous avons envisagé deux extensions au problème du RCPSP dans lesquelles nous définissons des contraintes additionnelles originales et nous avons vu qu'il est possible d'adapter les algorithmes pour prendre en compte ces contraintes. Nous proposons dans les deux sections suivantes de résoudre un problème de placement en considérant la relaxation de certaines contraintes, menant ainsi à un RCPSP. Nous montrons ainsi comment les algorithmes pour le RCPSP peuvent entrer en jeu dans la résolution d'un problème de placement.

2.5 Proposition d'un *double SGS* (*Schedule Generation Scheme*) pour un problème de placement

Dans cette section, nous proposons une méthode de résolution originale pour un problème de placement orthogonal en deux dimensions : le *two orthogonal packing problem* (2OPP). L'idée est de relaxer le problème pour se ramener à un RCPSP puis de le résoudre en deux étapes :

étape 1 une solution est calculée pour le problème relaxé ;

étape 2 la solution du problème relaxé est utilisée pour trouver une solution au problème de packing initial.

Nous présentons tout d'abord les problèmes de placement dans un cadre général. Nous établissons, ensuite, le lien entre la forme relaxée d'un problème de placement orthogonal en deux dimensions et le RCPSP. Enfin, nous décrivons la méthode de résolution.

2.5.1 Les problèmes de placement en deux dimensions

2.5.1.1 Présentation générale

Les problèmes de placement (*Packing Problems*) forment une classe de problèmes d'optimisation combinatoire importante. D'une manière générale, ils consistent à placer des objets dans des conteneurs qui peuvent être de dimension un, deux ou trois. Nous nous intéressons, ici, plus particulièrement aux problèmes en 2 dimensions, dans lesquels les conteneurs et les objets sont des rectangles de largeur et longueur données. Les problèmes

de placement en deux dimensions diffèrent principalement par leur fonction objectif. Parmi ceux-ci on trouve :

le *two dimensional bin packing problem* dans lequel, on dispose d'un nombre illimité de conteneurs identiques pour placer les objets. L'objectif est d'utiliser le moins de conteneurs possible pour placer tous les objets ;

le *strip packing problem* dans lequel, on dispose d'un unique conteneur de largeur fixe et de longueur infinie. L'objectif est de placer tous les objets de manière à ce que la longueur du conteneur utilisée soit la plus petite possible ;

le *two orthogonal packing problem* dans lequel, on dispose d'un unique conteneur de taille donnée. L'objectif est de déterminer si tous les objets peuvent être placés dans le conteneur. C'est sur ce problème que nous nous focalisons.

2.5.1.2 Le *two orthogonal packing problem*

Dans le *two orthogonal packing problem* (2OPP), on dispose d'un ensemble d'objets et on souhaite savoir s'ils peuvent être placés dans un conteneur de taille donnée. Les objets et le conteneur étant modélisés par des rectangles. Ce problème est *NP-complet* ([GJ79]).

Une instance est définie par un ensemble B d'objets b_i, $i = 1 \dots n$, et par un conteneur C de largeur l_C et longueur L_C. Chaque objet b_i possède une largeur l_i et une longueur L_i. On suppose que les objets ne peuvent pas subir de rotation. Le problème consiste alors à déterminer s'il est possible de placer tous les objets dans le conteneur sans qu'ils se superposent. Une solution est entièrement définie par la position $(x_i,\ y_i)$ du coin inférieur gauche de chaque objet. Un exemple d'instance et de solution est donné figure 2.16.

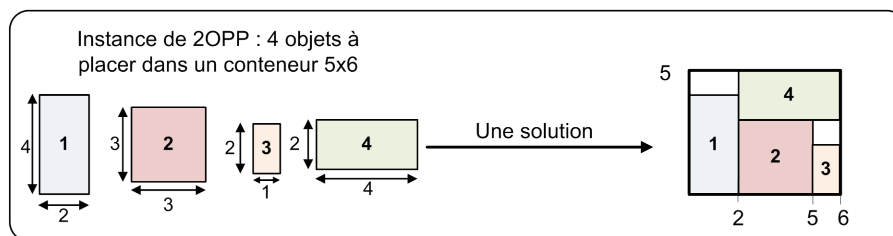


Fig. 2.16 – Exemple d'instance et solution pour le 2OPP

Diverses méthodes de résolution, exactes et approchées, existent pour ce problème. Parmi les méthodes heuristiques, on peut citer [LCT03] qui utilise une métaheuristique de type recuit simulé ou encore [Gon07] qui propose un algorithme génétique.

Parmi les méthodes exactes, on peut citer [MV98] qui propose un *branch and bound*. Leur méthode est reprise et améliorée dans [CCM07]. Cet article propose également une approche similaire à la notre, qui repose sur une relaxation du problème. Dans cet article la relaxation consiste à découper chaque objet de dimension $(L_i \times l_i)$ en l_i bandes de dimension $(L_i \times 1)$. Le problème relaxé consiste alors à trouver un placement pour les bandes. Dans ce placement, les bandes correspondant à un même objet possèdent la même abscisse mais ne sont pas forcément consécutives. Cette première étape est résolue par un *branch and bound*. Pour chaque solution trouvée un second *branch and bound* est exécuté pour

résoudre le problème initial. Dans cette seconde étape, on conserve les abscisses trouvées précédemment et on cherche à rendre toutes les bandes d'un même objet consécutives.

2.5.2 Le lien entre les problèmes de placement et le RCPSP

D'un point de vue applicatif, la famille des problèmes d'ordonnancement de projet à laquelle appartient le RCPSP est totalement différente de la famille des problèmes de placement. Cependant, il existe de nombreuses similitudes entre ces problèmes. Dans [Har99] et [Har00] Hartmann établit le lien théorique entre des problèmes de placement et des extensions du RCPSP. Il montre, par exemple, comment transformer un strip packing en un RCPSP multi-modes.

Notre approche est différente car nous considérons une relaxation du problème de placement de manière à obtenir un RCPSP classique, plus simple à résoudre. Nous utilisons, ensuite, la solution du problème relaxé pour trouver une solution au problème initial.

2.5.2.1 RCPSP et 2OPP

Dans un RCPSP, on considère un ensemble d'activités et on cherche leur date de début de manière à les planifier dans le cadre d'un projet. Dans un 2OPP on considère un ensemble d'objets rectangulaires et on cherche à les positionner dans un conteneur. Le tableau 2.13 donne les points communs et les différences entre les deux problèmes.

	2OPP	RCPSP
Définition du problème	Placer des petits rectangles (objets) dans un grand rectangle	Planifier des activités dans le temps avec des contraintes de ressources
Objets manipulés	Objets 2D définis par leurs dimensions	Activités définies par leur consommation de ressources et leur durée
Vocabulaire	Objet	Activité
	Largeur	Consommation
	Longueur	Durée
Solution	Positions (x,y) des objets	Dates de début des actions

Tab. 2.13 – Comparaison du 2OPP et du RCPSP

Dans une solution du RCPSP, les dates de début des activités peuvent être vues comme des abscisses dans un plan 2D, c'est d'ailleurs la démarche utilisée pour tracer un diagramme de Gantt (voir section 2.1.4.2). On constate alors qu'un 2OPP, dans lequel on autorise la préemption dans le sens de la largeur, peut être considéré comme un RCPSP à une ressource : un objet b_i du 2OPP de longueur L_i et de largeur l_i peut être considéré comme une activité du RCPSP de durée $d_i = L_i$ et de consommation $r_i = l_i$. La largeur du conteneur constitue la quantité de ressources disponible et sa longueur représente le makespan à ne pas dépasser (voir figure 2.17). Une solution du RCPSP donne alors la date de début x_i des activités c'est-à-dire l'abscisse des objets associés. De plus, comme à chaque instant la solution respecte les contraintes de ressources, on est assuré, qu'en chaque

abscisse x, la somme des largeurs des objets en x ne dépasse pas la largeur du conteneur.

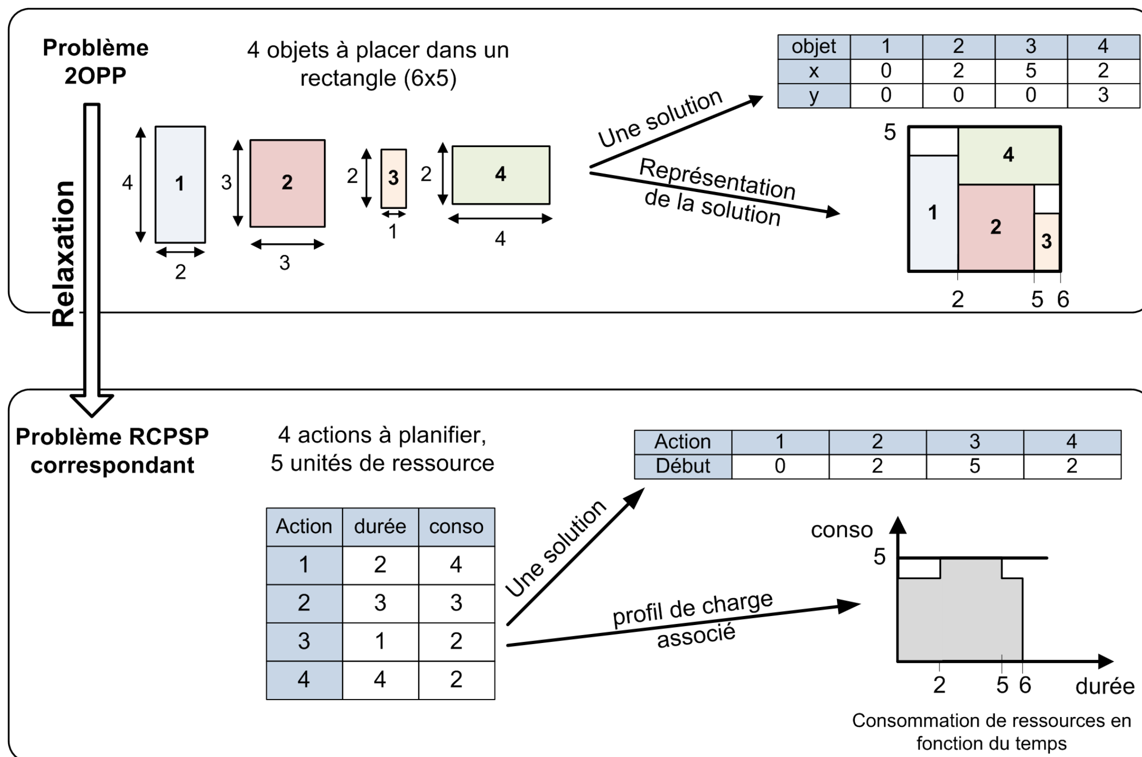


FIG. 2.17 – Exemple d'instance du 2OPP transformée en instance du RCPSP

Cependant, rien ne garantit qu'il soit possible de représenter les objets (rectangles) dans un plan en deux dimensions de manière à respecter les abscisses x_i et sans préemption. En effet, une solution du RCPSP ne donne aucune indication sur les ordonnées des objets associés aux activités donc sa représentation en 2D (sous forme de diagramme de Gantt par exemple) peut induire une préemption sur les objets dans le sens de la largeur comme le montre la figure 2.18. Cette figure représente la solution du RCPSP du tableau 2.14 associée à un conteneur de largeur 40 et longueur 20. Il est possible de vérifier (par programmation linéaire par exemple) qu'il n'existe aucune solution sans préemption respectant les abscisses données par le tableau 2.14. Cependant, ce cas de figure arrive très rarement et il existe presque toujours un placement qui respecte les abscisses données par le RCPSP. A partir de cette constatation, on peut construire une méthode de résolution du 2OPP qui calcule tout d'abord les abscisses grâce à la relaxation en RCPSP, puis les ordonnées de manière à obtenir un placement sans préemption. Cette méthode est détaillée dans la section suivante.

activités	1	2	3	4	5	6	7	8	9	10	11
durée	6	8	5	2	13	5	12	2	7	2	3
conso.	9	15	10	22	7	11	6	29	13	31	13
x_i	2	0	2	0	7	10	2	16	8	18	8

TAB. 2.14 – Exemple d'instance pour le RCPSP à 11 activités, un seul type de ressources (disponibles en quantité 40) et solution (début x_i des actions) associée

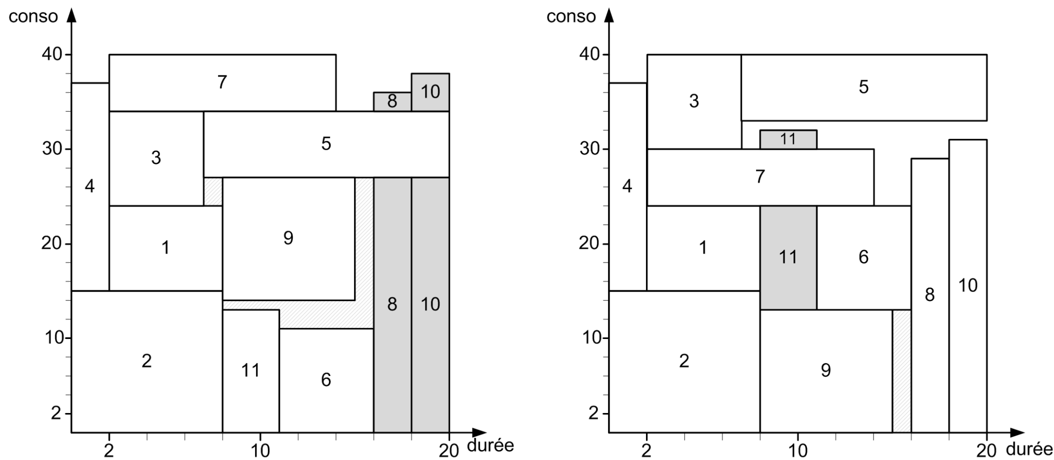


FIG. 2.18 – Deux diagrammes de Gantt associés à la solution du tableau 2.14, au moins un rectangle est coupé

2.5.3 Résolution du 2OPP

Comme on l'a vu précédemment on résout le 2OPP en deux étapes :

étape 1 : on résout le RCPSP associé au problème de 2OPP de manière à obtenir les abscisses des objets ;

étape 2 : on résout le 2OPP en imposant l'abscisse des objets calculée à l'étape 1.

L'intérêt de cette résolution en deux étapes est clairement de réduire l'espace de recherche. En effet, on cherche à chaque étape à fixer une seule des deux dimensions, ce qui limite l'explosion combinatoire des placements possibles.

2.5.3.1 Résolution du RCPSP (étape 1)

On utilise pour résoudre le RCPSP un schéma de génération d'ordonnancements (voir section 2.1.5.1). Comme le RCPSP qui dérive du 2OPP est très peu contraint (un seul type de ressource et aucune contrainte de précédence), il est inutile d'utiliser des règles de priorité. En effet, ces dernières étant calculées, en grande partie, grâce aux contraintes de précédence et à l'utilisation simultanée de différents types de ressources elles donnent, dans la configuration qui nous intéresse, des résultats médiocres. On choisit donc à chaque itération l'activité à insérer de manière aléatoire. De plus, on doit s'assurer de ne pas dépasser la longueur du conteneur L_C. Le schéma de génération d'ordonnancements est donc modifié comme le montre l'algorithme 28.

2.5.3.2 Résolution du 2OPP à abscisses fixées (étape 2)

La résolution du 2OPP à abscisses fixées est assurée par l'algorithme `Placement`. L'idée est d'empiler les objets en respectant leur position en abscisse. Ce problème est résolu avec un algorithme glouton similaire au précédent. Initialement, aucun objet n'est placé dans le conteneur. A chaque itération, on place un nouvel objet.

Algorithme 28 : SGS_RCPSP

Entrées :
- $A = \{1, \ldots, n\}$ //ensemble des activités j de durée d_j et consommation r_j,
- L_C //makespan à ne pas dépasser (longueur du conteneur)

Sorties :
- $reussite$ //$reussite = 1$ si on a trouvé un makespan plus petit ou égal à L_C, 0 sinon
- $\{ES_j\}_{j=1 \ldots n}$ //date de début au plus tôt de j
- $\{LS_j\}_{j=1 \ldots n}$ //date de début au plus tard de j

```
1  t ← 0 ;
2  Ac ← ∅ //ensemble des activités actives en t ;
3  D ← A //ensemble des activités insérables en t ;
4  k ← 1;
5  Tant que k ≤ n faire
6  |  Choisir aléatoirement dans D une activité j ;
7  |  ES_j = t ;
8  |  Mettre à jour les ensembles Ac et D ;
9  |  Tant que D = ∅ faire
10 |  |  t ← min_{j∈Ac} (ES_j + d_j);
11 |  |  Mettre à jour les ensembles Ac et D ;
12 |  k = k + 1;
13 C_max ← max_{j∈A} (ES_j + d_j) ;
14 Si C_max > L_C alors
15 |  reussite ← faux ;
16 sinon
17 |  reussite ← vrai ;
18 |  Calculer les LS_j ;
```

On définit, pour chaque itération de l'algorithme `Placement` :

y_{min} : la plus petite ordonnée à laquelle un nouvel objet peut être placé compte tenu des objets déjà placés ;

D : l'ensemble des objets i qui peuvent être placés en (x_i, y_{min}).

L'algorithme est initialisé avec $y_{min} = 0$ et $D = B$ (où B est l'ensemble des objets à placer). A chaque itération, un objet b_i de D est choisi aléatoirement et positionné en (x_i, y_{min}). La valeur de y_{min} est alors recalculée pour prendre en compte cette nouvelle insertion et l'ensemble D est mis à jour. On continue itérativement jusqu'à avoir placé tous les objets ou jusqu'à ce qu'il ne soit plus possible d'insérer un nouvel objet sans dépasser la largeur du conteneur (voir algorithme 29).

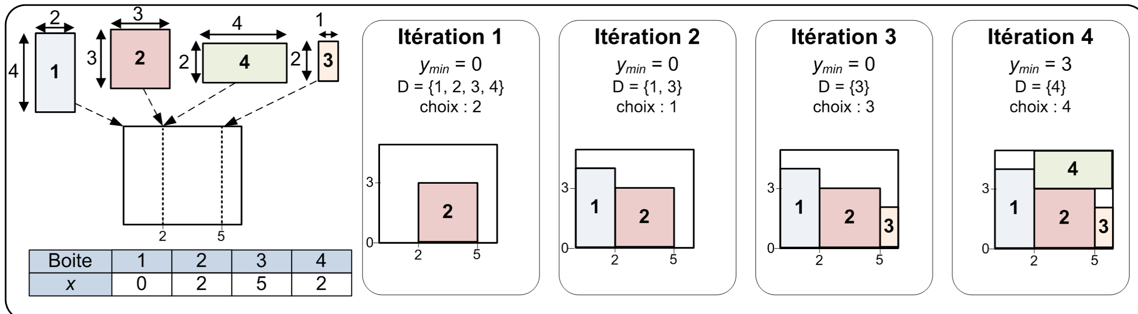
La figure 2.19 illustre l'algorithme `Placement` sur un problème à quatre objets.

Algorithme 29 : Placement

Entrées :
- $B = \{b_1 \dots b_n\}$ //ensemble des objets b_i de largeur l_i et longueur L_i,
- $\{x_i\}_{i=1 \dots n}$ //x_i abscisse de l'objet b_i,
- l_C //largeur du conteneur

Sorties :
- *reussite* //*reussite* = 1 si on a trouvé un placement sans dépasser l_C, 0 sinon
- $\{y_1 \dots y_n\}_{i=1 \dots n}$ //y_i ordonnée de l'objet b_i

```
1  reussite ← vrai ;
2  y_min ← 0 ;
3  Ac ← ∅ //ensemble des objets i déjà placés et tels y_i ≤ y_min < y_i + l_i;
4  D ← B //ensemble des objets que l'on peut placer en y_min;
5  stop ← faux ;
6  k ← 1
7  Tant que k ≤ n et stop = faux faire
8      Choisir aléatoirement b_i dans D ;
9      y_i ← y_min ;
10     Mettre à jour les ensembles Ac et D ;
11     Tant que D = ∅ faire
12         y_min ← min_{i∈Ac} (y_i + l_i) ;
13         Mettre à jour les ensembles Ac et D ;
14     Si max_{i non inséré} (y_min + l_i) > l_C alors
15         stop ← vrai ;
16         reussite ← faux ;
17     k ← k + 1 ;
```



Boite	1	2	3	4
x	0	2	5	2

FIG. 2.19 – Illustration de l'algorithme de placement : l'objet 1 doit être placé à l'abscisse 0, les objets 2 et 4 à l'abscisse 2 et l'objet 3 à l'abscisse 5

2.5.3.3 Algorithme complet de résolution du 2OPP

L'algorithme complet de résolution du 2OPP est donné par l'algorithme 30. A l'étape 1, on exécute l'heuristique `SGS_RCPSP` jusqu'à ce qu'elle donne une solution qui respecte la longueur du conteneur ou jusqu'à ce qu'un nombre maximal d'itérations soit atteint. Lorsque cette première étape donne une solution, on exécute l'algorithme de placement. L'abscisse d'un objet b_i est alors fixée successivement à la date de début au plus tôt (ES_i) de l'activité i correspondante puis à sa date de début au plus tard (LS_i). L'algorithme s'arrête

dès qu'un placement est trouvé ou qu'un nombre maximal d'itérations est atteint.

Le fait de choisir les dates de début au plus tôt, puis au plus tard, comme abscisse permet de rechercher plus de positions pour les objets. On augmente ainsi les chances de trouver un placement.

Algorithme 30 : DoubleSGS

Entrées : $B = \{b_1 \dots b_n\}$, l_C, L_C, $itMax1$, $itMax2$
Sorties : $reussiteOPP$, $X = \{x_1 \dots x_n\}$, $Y = \{y_1 \dots y_n\}$

```
1  Construire l'ensemble d'activités A correspondant aux objets de B : pour chaque
   objet b_i de longueur L_i et largeur l_i, on crée l'activité i de durée d_i = L_i et de
   consommation r_i = l_i. La ressource est disponible en quantité M = L_C;
2  i ← 0;
3  Tant que reussiteOPP = faux et i < maxRepet faire
4      i ← i + 1;
5      //Etape 1 :
6      k ← 1;
7      Répéter
8          (reussiteRCPSP, {ES_j}, {LS_j}) = SGS_RCPSP(A, L_C) ;
9          k ← k + 1;
10     jusqu'à reussiteRCPSP = vrai ou k > itMax1 ;
11     //Etape 2 :
12     Si reussiteRCPSP = vrai alors
13         k ← 1;
14         Répéter
15             X = {ES_j} ;
16             (reussiteOPP, Y) = Placement(B, X, l_C) ;
17             Si reussiteOPP = faux alors
18                 X = {LS_j} ;
19                 (reussiteOPP, Y) = Placement(B, X, l_C) ;
20             k ← k + 1;
21         jusqu'à reussiteOPP = vrai ou k > itMax2 ;
22     sinon
23         reussiteOPP ← faux ;
```

Remarque 6
On peut remarquer qu'il est facile de calculer une borne inférieure. En effet, on peut considérer les activités incompatibles : qui ne peuvent pas être planifier à la même date à cause des contraintes de ressource. Il s'agit de la LB3 de Klein et Scholl [KS99b]. Cette borne inférieure est très rapide à calculer.

2.5.4 Résolution du 2OPP avec rotations

On considère, ici, le 2OPP avec l'hypothèse supplémentaire que les rotations d'objets sont autorisées. On peut alors choisir de placer un objet b_i de dimension $L_i \times l_i$ dans le sens initial $L_i \times l_i$ ou dans le sens $l_i \times L_i$ (on dit alors que l'objet a subit une rotation).

Pour résoudre ce problème, on procède de la même manière que dans le cas classique. La rotation est prise en compte dans l'étape 1 : les activités sont dupliquées de sorte que chaque activité i de durée d_i et consommation r_i possède une activité homologue i' de durée $d_{i'} = r_i$ et consommation $r_{i'} = d_i$. Les activités i telles que la durée dépasse la largeur du conteneur ($d_i > l_C$) ou la consommation dépasse la longueur du conteneur ($r_i > l_c$) ne peuvent pas être dupliquées. On résout alors le RCPSP avec l'ensemble des activités A' qui contient à la fois les activités originales et les activités homologues. L'algorithme `SGS_RCPSP` est légèrement modifié pour s'assurer que parmi l'activité originale et l'activité homologue une et une seule est planifiée : après avoir choisi aléatoirement une activité j dans D (ligne 6), on supprime son homologue. A l'étape 2, on applique simplement l'algorithme `Placement` en considérant les objets dans le sens choisi à l'étape 1.

2.5.5 Résultats numériques

On utilise le jeu d'instances proposé par Clautiaux *et al.* ([CJCM08]). Il est composée de 41 instances comportant de 10 à 23 objets. Sur ces 41 instances 15 sont réalisables. Plusieurs méthodes de la littérature ont déjà été testées sur ces instances :

- OPP de Fekete *et al.* ([FSdV07]) ;
- TSBP ([CCM07]) et ER ([CJCM08]) de Clautiaux *et al.* ;
- SWEEP de Beldiceanu et Carlsson ([BC01]).

Le *double SGS* est programmé en C++. Les tests ont été réalisés sur un PC AMD Opteron 2.1GHz sous Linux. Clautiaux *et al.* utilise un Pentium M 1.8GHz sous windows XP pour la méthode ER, et un Pentium IV 2.6 GHz pour la méthode TSBP. Pour la méthode OPP un Pentium IV 3.2 GHz a été utilisé. En ce qui concerne la méthode SWEEP, elle utilise SICStus Prolog (`http://www.sics.se/isl/sicstuswww/site/index.html`).

Les résultats sont synthétisés dans le tableau 2.15. Le colonne « Instances » donne le nom de l'instance, le nombre n d'objets à placer et la faisabilité de l'instance. La colonne « Méthodes exactes » donne le temps cpu en secondes mis par la méthode concernée pour déterminer si l'instance est faisable ou pas. Le symbole « - » signifie que la méthode n'a pas trouvée de réponse dans le temps imparti (15 minutes pour OPP et SWEEP, 4 minutes pour TSBP).

Le *double SGS* est exécuté avec les paramètres suivants : $itMax1 = 1\ 000 \times n$, $itMax2 = 100 \times n$ et $maxRepet = 5$. Il est exécuté cinq fois. La colonne correspondante est divisée en quatre sous-colonnes :

- la colonne « RCPSP » contient *non* si aucune solution au problème du RCPSP associé au 2OPP n'a été trouvée. Elle contient *oui* si une solution a été trouvée au moins une fois. Dans ce cas on indique sur les 5 exécutions combien ont mené à une solution.
- la colonne « réal. » contient *non* si aucune solution n'a été trouvée. Elle contient *oui* si une solution a été trouvée au moins une fois. Dans ce cas on indique sur les 5 exécutions combien ont mené à une solution. Le symbole « * » signifie que l'heuristique fournit la bonne réponse ;
- la colonne « cpu moy. » donne le temps cpu moyen en secondes d'une exécution ;
- la colonne « best cpu » donne le meilleur temps cpu sur les cinq exécutions.

Instances			Méthodes exactes				Heuristique				
			OPP	TSBP	SWEEP	ER	double SGS				
nom	n	réal.	[FSdV07]	[CCM07]	[BC01]	[CJCM08]	cette thèse				
			cpu	cpu	cpu	cpu	RCPSP	réal.		cpu moy.	best cpu
E00N23	23	non	-	289	-	6.75	non	non	*	0.72	0.71
E00X23	23	non	-	86	-	1.20	non	non	*	0.71	0.70
E05N15	15	non	0	0	-	0.01	non	non	*	0.42	0.42
E05X15	15	non	2	0	17.03	0.20	non	non	*	0.42	0.41
E05F20	20	oui	491	2	0.00	1.63	oui(5/5)	oui(5/5)	*	0.04	0.00
E04F20	20	oui	22	3	11.96	1.40	oui(5/5)	oui(5/5)	*	0.01	0.00
E13N15	15	non	0	0	0.08	0.01	non	non	*	0.42	0.41
E10N15	15	non	0	0	-	0.06	non	non	*	0.44	0.43
E03N16	16	non	2	32	62.90	1.02	non	non	*	0.47	0.46
E20F15	15	oui	0	1	0.01	0.60	oui(5/5)	oui(5/5)	*	0.00	0.00
E04N15	15	non	0	1	11.04	0.79	non	non	*	0.43	0.43
E03N15	15	non	0	1	1.16	0.72	non	non	*	0.43	0.43
E10X15	15	non	0	1	6.85	0.46	non	non	*	0.43	0.43
E07N15	15	non	0	1	10.39	0.59	non	non	*	0.42	0.41
E05N17	17	non	0	1	11.23	0.37	non	non	*	0.49	0.48
E08N15	15	non	0	1	2.11	0.33	non	non	*	0.42	0.41
E07F15	15	oui	0	1	1.88	0.35	oui(5/5)	oui(5/5)	*	0.00	0.00
E03X18	18	oui	0	22	0.33	0.55	oui(5/5)	oui(5/5)	*	0.15	0.02
E04N18	18	non	10	7	329.21	0.31	non	non	*	0.51	0.50
E02F20	20	oui	-	12	42.25	0.84	oui(4/5)	oui(3/5)	*	0.46	0.10
E04F17	17	oui	13	26	0.00	0.54	oui(5/5)	oui(5/5)	*	0.04	0.00
E13N10	10	non	0	0	0.51	0.16	non	non	*	0.26	0.25
E04F15	15	oui	0	1	1.29	0.43	oui(3/5)	oui(3/5)	*	0.35	0.18
E03N17	17	non	0	4	0.89	0.40	non	non	*	0.47	0.46
E00N15	15	non	0	2	136.59	0.44	non	non	*	0.43	0.42
E20X15	15	oui	0	44	0.00	0.24	oui(5/5)	non		0.00	0.00
E05F15	15	oui	0	3	0.27	0.30	oui(5/5)	oui(5/5)	*	0.01	0.00
E02F17	17	oui	7	12	0.97	0.52	non	non		0.51	0.50
E02F22	22	oui	167	4	0.38	0.79	oui(5/5)	oui(5/5)	*	0.03	0.00
E04F19	19	oui	560	7	0.08	0.77	oui(5/5)	oui(5/5)	*	0.01	0.00
E05F18	18	oui	0	126	0.02	0.51	oui(5/5)	oui(5/5)	*	0.06	0.01
E08F15	15	oui	0	117	0.00	0.38	oui(5/5)	oui(5/5)	*	0.01	0.00
E04N17	17	non	0	1	2.31	0.23	non	non	*	0.48	0.46
E13X15	15	non	0	0	9.34	0.07	non	non	*	0.40	0.39
E15N15	15	non	0	0	6.78	0.17	non	non	*	0.42	0.41
E00N10	10	non	0	0	0.04	0.01	non	non	*	0.28	0.27
E02N20	20	non	0	1	2.00	0.32	non	non	*	0.58	0.57
E03N10	10	non	0	0	0.00	0.09	non	non	*	0.27	0.27
E07N10	10	non	0	0	0.19	0.17	non	non	*	0.27	0.27
E07X15	15	non	0	0	0.06	0.00	non	non	*	0.42	0.41
E10N10	10	non	0	0	0.02	0.05	non	non	*	0.27	0.27
moyenne			33.53	19.73	18.11	0.61				0.32	0.29

TAB. 2.15 – Comparaison des résultats de notre méthode *double SGS* avec ceux de la littérature (tableau extrait de [CJCM08] et mis à jour)

On constate tout d'abord que l'existence d'une solution pour le 2OPP relaxé en RCPSP donne un très bonne indication sur l'existence d'une solution au 2OPP initial. En effet, sur les 24 instances non réalisables pour le 2OPP on remarque qu'il n'y en a aucune pour laquelle une solution au problème relaxé a été trouvé.

On constate ensuite que notre méthode est efficace : elle donne 13 solutions sur les 15 instances réalisables. De plus, parmi celles-ci, 11 instances sont résolues à chaque exécution de l'algorithme. Les temps de calcul sont très faibles : 0.32 secondes en moyenne soit deux fois plus faibles que la plus rapide des méthodes exactes et environ 100 fois plus faibles que la plus lente.

Notons que les instances proposées par Clautiaux ont un nombre limité d'objets (23 au maximum) mais notre schéma heuristique permettrait de traiter des instances de beaucoup plus grande taille.

2.5.6 Conclusion

Dans cette section, nous proposons une nouvelle approche pour résoudre un problème de placement orthogonal en deux dimensions (le *Two Orthogonal Packing Problem* - 2OPP). Le 2OPP consiste à placer des petits rectangles (objets) dans un grand rectangle (conteneur) sans qu'ils se superposent. Le but est alors de déterminer si oui ou non les objets peuvent être placés dans le conteneur et, dans ce cas, de trouver les coordonnées des objets. Nous proposons une résolution originale en deux temps. Dans un premier temps, le problème est relaxé en un RCPSP. Nous déduisons de la résolution du RCPSP les abscisses des objets. Dans un deuxième temps, nous utilisons ces abscisses pour résoudre le problème initial grâce à un algorithme d'insertion. Cette approche s'avère très efficace et très rapide. Elle a été testée sur des instances de la littérature et comparée à quatre méthodes existantes. Bien qu'elle soit heuristique, elle fournit presque toujours une réponse juste au problème. Elle est la méthode qui donne les plus faibles temps de calcul.

Cette méthode a également été utilisée dans le cadre d'un problème de tournées de véhicules avec contraintes de chargement en deux dimensions (le 2L-CVRP, voir chapitre 4) avec et sans rotation. Elle s'avère également très efficace dans ce problème et permet d'obtenir de meilleurs résultats que ceux de la littérature.

2.6 Proposition d'un *double flot* pour un problème de placement

Dans cette section, nous présentons une seconde méthode pour résoudre le 2OPP. Elle repose sur les même idées que la méthode *double SGS* de la section précédente, à savoir sur la relaxation du problème de placement en RCPSP. La différence vient du fait qu'on utilise une modélisation à l'aide d'un multiflot particulier pour résoudre le problème au lieu de déterminer directement les coordonnées des objets. Nous commençons par donner le principe général de cette modélisation. Nous démontrons ensuite l'équivalence entre un multiflot possédant certaines propriétés et une solution du 2OPP. Enfin, nous proposons un algorithme pour construire ce multiflot et fournissons les résultats des expérimentations numériques sur les instances de Clautiaux *et al.* ([CJCM08]).

2.6.1 Principe général

Cette nouvelle méthode, que l'on nomme *double flot*, repose sur le schéma algorithmique suivant :

Etape 1 : Le problème de 2OPP est relaxé en RCPSP. Le RCPSP est résolu de sorte que les échanges de ressources entre activités s'expriment à l'aide d'un flot (voir section 2.2). Ce flot donne une indication sur la position relative (de gauche à droite) des objets auxquels sont associées les activités ;

Etape 2 : A partir du flot calculé à l'étape 1, un nouveau flot est calculé pour connaître la position relative des objets de bas en haut. A partir de ces deux flots, on peut déduire la position exacte des objets.

Nous proposons donc de définir un graphe représentant une solution du 2OPP, comme un graphe dans lequel circule deux flots :
- le premier flot représente les contraintes entre les abscisses des objets ;
- le deuxième représente les contraintes entre les ordonnées des objets.

La figure 2.20 illustre un tel graphe et le placement associé. On représente dans le graphe uniquement les arcs portant du flot.

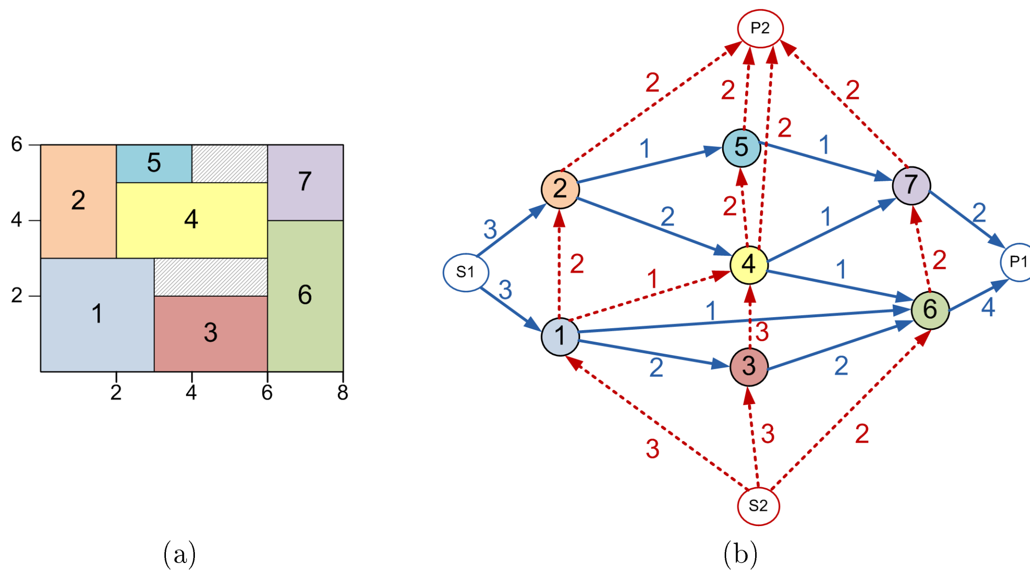


Fig. 2.20 – (a) Placement de 7 objets ; (b) double flot associé à ce placement : en traits pleins le flot obtenu à l'étape 1 et en traits pointillés le flot obtenu à l'étape 2

Sur la figure 2.20(b), le flot en traits pleins représente la solution du RCPSP obtenue à l'étape 1, c'est-à-dire l'échange de ressources entre les activités (la correspondance entre une solution du RCPSP et un multiflot a été présentée section 2.2).

On peut déduire de ce flot la position relative de gauche à droite des objets. En effet, si une activité i transfère du flot à une activité j, alors i se termine forcément avant le début de j. Dans le placement correspondant, on en déduit que l'objet b_i (associé à l'activité i) est placé avant l'objet b_j (associé à l'activité j). Ainsi, on déduit du flot de la figure 2.20(b) que 2 est avant 4 et 5, 1 est avant 3 et 6, 4 est avant 7 et 6 etc.

Les flots en traits pointillés représentent le deuxième flot. De même que pour le RCPSP, on ajoute une source (S2) et un puits (P2) à l'ensemble des activités. Si l'arc (i, j) porte du flot (en pointillés) de valeur f, cela signifie que dans le placement, i et j ont f abscisses en commun. De plus, sur la bande correspondant à ces f abscisses aucun autre objet ne

peut venir s'intercaler. Par exemple, considérons le flot en pointillés de 3 vers 4 de valeur 3. Il signifie que l'objet 4 est placé juste au-dessus de l'objet 3 et qu'il partage 3 abscisses. En effet, sur le placement on constate que 4 est juste au dessus de 3 aux abscisses 3, 4 et 5. Ce deuxième flot donne donc une indication sur la position relative des objets de bas en haut.

Notons que, contrairement à la méthode *double SGS* de la section précédente, le double flot ne fournit pas les abscisses et les ordonnées des objets mais simplement leurs positions relatives. Il faut ensuite interpréter ce résultat pour obtenir les coordonnées des objets. La section suivante établit l'équivalence entre une solution du 2OPP et un multiflot possédant certaines propriétés.

2.6.2 Relation entre double flot et solution du 2OPP

2.6.2.1 Définition du 2OPP

Une instance de 2OPP est définie par un ensemble B d'objets b_i, $i = 1 \ldots n$, et par un conteneur C de largeur l_C et longueur L_C. Chaque objet b_i possède une largeur l_i et une longueur L_i. On suppose que les objets ne peuvent pas subir de rotation. Le problème consiste alors à déterminer s'il est possible de placer tous les objets dans le conteneur sans qu'ils se superposent. Une solution est entièrement définie par la position $(x_i,\ y_i)$ du coin inférieur gauche de chaque objet. On suppose, par la suite, que les coordonnées du coin inférieur gauche du conteneur sont (0,0).

2.6.2.2 Calcul du double flot à partir d'une solution du 2OPP

Etant donné une solution du 2OPP, on définit un graphe complet $G = (X, E)$ dans lequel l'ensemble des sommets $X = \{1, \ldots, n\}$ représente l'ensemble des objets B. Un sommet i est associé à un objet b_i. On ajoute à X deux sommets fictifs source (s) et puits (p) et on note X^* l'ensemble des sommets $X \cup \{s\} \cup \{p\}$. Afin d'alléger les notations, on associe à un sommet i de X la longueur L_i et la largeur l_i de l'objet b_i correspondant ainsi que sa position (x_i, y_i) dans la solution.

Afin de formaliser la manière dont sont calculés les flots à partir d'une solution, on introduit quelques notations. On note $S = [(x_{s1}, y_{s1}); (x_{s2}, y_{s2})]$ le segment dans le plan $\mathbb{R}^2$ dont la première extrémité se situe au point de coordonnées (x_{s1}, y_{s1}) et la seconde au point de coordonnées (x_{s2}, y_{s2}). On dit qu'un segment $S = [(x_{s1}, y_{s1}); (x_{s2}, y_{s2})]$ intersecte un objet b_i s'il a au moins un point en commun avec la surface $[(x_i, y_i); (x_i + L_i, y_i)] \times [(x_i, y_i); (x_i, y_i + l_i)[$. Notons que les points d'ordonnée $y_i + l_i$ ne sont pas pris en compte.

On introduit, de plus, deux ensembles I, J associés à un sommet i et définis par :

$I_i = \{t \in [0, l_C]$ tel que le segment $[(0, t); (x_i, t)]$ intersecte b_i et seulement $b_i\}$;

$J_i = \{t \in [0, l_C]$ tel que le segment $[(x_i + L_i, t); (L_C, t)]$ intersecte b_i et seulement $b_i\}$.

Un troisième ensemble K_{ij} est associé à deux sommets i et j :

$K_{ij} = \{t \in [0, l_C]$ tel que le segment $[(x_i + L_i, t); (x_j, t)]$ intersecte i et j et n'intersecte aucun objet autre que i et $j\}$

Le flot $\mathcal{F}$, qui exprime la position relative des objets de gauche à droite, se définit alors de la manière suivante :

- $\forall i \in X$ $\mathcal{F}_{s,i} = |I_i|$;
- $\forall i \in X$ $\mathcal{F}_{i,p} = |J_i|$;
- $\forall i \in X$, $j \in X$, $\mathcal{F}_{i,j} = |K_{ij}|$.

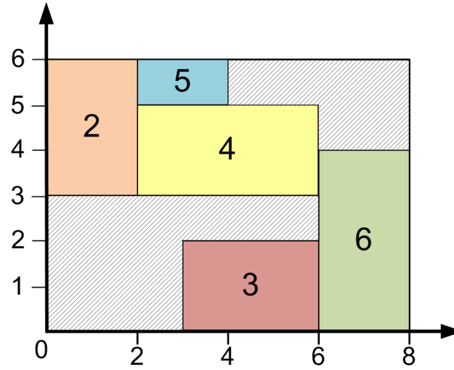


FIG. 2.21 – Exemple de placement

Par exemple sur la figure 2.21, les ensembles non vides associés à 4 sont : $J_4 = \{4\}$, $K_{2,4} = \{3, 4\}$ et $K_{4,6} = \{3\}$. On en déduit que les valeurs non nulles de flot relatifs à 4 sont $\mathcal{F}_{4,p} = |J_4| = 1$, $\mathcal{F}_{2,4} = |K_{2,4}| = 2$ et $\mathcal{F}_{4,6} = |K_{4,6}| = 1$.

Le flot $\mathcal{G}$, qui exprime la position relative des objets de bas en haut, se définit de la même manière que $\mathcal{F}$ en échangeant les longueurs et les largeurs.

Intuitivement, le flot $\mathcal{F}$ transporte la « ressource géométrique » constituée par la largeur du conteneur. Cette ressource est transmise entre les sommets de la même manière qu'une ressource classique dans le RCPSP. Le flot $\mathcal{G}$ transporte la « ressource géométrique » constituée par la longueur du conteneur.

Le multiflot défini par le flot $\mathcal{F}$ et le flot $\mathcal{G}$ est appelé double flot.

2.6.2.3 Propriétés du double flot

Propriété 1 : Pour tout sommet i, j de X, si $\mathcal{F}_{i,j}$ est non nul alors $\mathcal{G}_{i,j}$ est nul et vice versa. En effet, si les deux quantités $\mathcal{F}_{i,j}$ et $\mathcal{G}_{i,j}$ sont non nuls cela signifie que j et i ont au moins une ordonnée en commun et une abscisse en commun, ce qui est impossible puisqu'ils ne se superposent pas.

Soit $G_{\mathcal{F}\cup\mathcal{G}} = (X^*, E_{\mathcal{F}\cup\mathcal{G}})$. L'ensemble des arcs $E_{\mathcal{F}\cup\mathcal{G}}$ est tel que : il existe un arc orienté de i vers j si $\mathcal{F}_{i,j} + \mathcal{G}_{i,j} \neq 0$. Soit $G_{\mathcal{F}\cup\mathcal{G}^-} = (X^*, E_{\mathcal{F}\cup\mathcal{G}^-})$. L'ensemble des arcs $E_{\mathcal{F}\cup\mathcal{G}^-}$ est tel que : il existe un arc orienté de i vers j si $\mathcal{F}_{i,j} + \mathcal{G}_{j,i} \neq 0$.

Propriété 2 : aucun des deux graphes $G_{\mathcal{F}\cup\mathcal{G}}$ et $G_{\mathcal{F}\cup\mathcal{G}^-}$ n'admet de circuit.

Démonstration de la propriété 2 :

On montre le résultat pour le graphe $G_{\mathcal{F}\cup\mathcal{G}}$, le résultat pour le graphe $G_{\mathcal{F}\cup\mathcal{G}^-}$ étant « symétrique ». En effet $\mathcal{G}^-$ représente le placement de haut en bas donc $G_{\mathcal{F}\cup\mathcal{G}^-}$ serait le graphe obtenu en retournant le conteneur.

On considère le graphe orienté $G^* = (X^*, E^*)$. L'arc (i,j) appartient à E^* s'il existe au moins un segment horizontal $[(x_i + L_i, t); (x_j, t)]$, $t \in \mathbb{N}$ ou un segment vertical $[(u, y_i + l_i); (u, y_j)]$, $u \in \mathbb{N}$ qui intersecte i et j. Notons que, contrairement à la définition du flot $\mathcal{F}$, on n'interdit pas l'intersection avec un autre objet. Il est clair que $G_{\mathcal{F}\cup\mathcal{G}}$ est inclut dans G^*. Donc pour montrer que $G_{\mathcal{F}\cup\mathcal{G}}$ est sans circuit, il suffit de montrer que G^* est sans circuit.

Supposons que G^* admette un circuit $\Gamma = (c_0, c_1, \ldots, c_p = c_0)$. Sans perte de généralité, on peut le choisir de longueur minimale. Tous les arcs de Γ ne peuvent pas porter du flot $\mathcal{F}$ (sinon il n'y a pas de circuit). On suppose que le premier arc (c_0, c_1) porte du flot $\mathcal{F}$ (il ne porte donc pas de flot $\mathcal{G}$). Soit $(c_g, c_{g'})$ le premier arc qui porte du flot $\mathcal{G}$ dans Γ. On considère les rectangles R_0, R_g et $R_{g'}$ respectivement associés aux sommets c_0, c_g et $c_{g'}$. A cause de la minimalité du circuit et comme la projection sur l'axe vertical de $\mathbb{R}^2$ du côté gauche de R_0 couvre tout l'intervalle défini par $y_0 \ldots y_0 + l_0$, il vient que $y_{g'}$ doit être au moins égal à $y_0 + l_0$ (sinon $\Gamma - \{c_g\}$ forme encore un circuit et de longueur plus petite que Γ, ce qui contredit l'hypothèse de minimalité). De même, en considérant la projection horizontale, on déduit que $x_{g'}$ doit être au moins égal à $x_0 + L_0$. De la même manière, on peut montrer que pour tout $h \geq g'$ on a $x_h \geq x_0 + L_0$ et $y_h \geq y_0 + l_0$. Il est donc impossible d'avoir un arc (c_h, c_0) car cela impliquerait $x_h + L_h \leq x_0$ ou $y_h + l_h \leq y_0$. On a donc une contradiction avec l'hypothèse initiale « Γ est un circuit ».

On a montré que G^* n'admet pas de circuit, donc $G_{\mathcal{F}\cup\mathcal{G}}$ n'admet pas de circuit. □

On dit qu'un double flot ($\mathcal{F}$, $\mathcal{G}$) est *sans circuit* si et seulement si les graphes $G_{\mathcal{F}\cup\mathcal{G}}$ et $G_{\mathcal{F}\cup\mathcal{G}^-}$ n'admettent pas de circuit.

2.6.2.4 Conversion d'un problème de 2OPP en un problème de flots

On dit qu'un double flot $(\mathcal{F}, \mathcal{G})$ est associé à une instance définie par un ensemble B d'objets et un conteneur C de dimension $L_C \times l_C$ si $\mathcal{F}$ transporte l_C unités de flot et $\mathcal{G}$ transporte L_C unités de flot.

Théorème 5
Une instance du 2OPP admet une solution si et seulement s'il existe un double flot sans circuit associé à cette instance.

Démonstration
On a déjà montré qu'une solution du 2OPP peut être représentée en double flot sans circuit. Il reste à montrer comment un double flot sans circuit peut être converti en solution du 2OPP. Il s'agit donc de déterminer les positions des objets à partir du double flot. Pour cela, on procède en trois étapes :

1. A partir du double flot sans circuit, on construit deux graphes orientés $G_x = (X^*, E_x)$ et $G_y = (X^*, E_y)$ de sorte qu'il existe dans $E_x \cup E_y$, soit un arc de i vers j, soit un

arc de j vers i ;

2. On donne un coût aux arcs de G_x et G_y de sorte que, par un calcul de plus longs chemins, on obtienne les positions du coin inférieur gauche des objets dans le placement ;
3. On s'assure que le placement obtenu respecte la longueur et la largeur du conteneur.

Rappel : $G_{\mathcal{F}} = (X^*, E_{\mathcal{F}})$ est le graphe orienté déduit du flot $\mathcal{F}$: il existe un arc (i, j) si $\mathcal{F}_{ij}$ est non nul. De la même manière $G_{\mathcal{G}} = (X, E_{\mathcal{G}})$ est le graphe orienté déduit du flot $\mathcal{G}$. Les graphes $G_{\mathcal{F}\cup\mathcal{G}}$ et $G_{\mathcal{F}\cup\mathcal{G}^-}$ sont sans circuit.

Etape 1

On construit E_x à partir de l'ensemble d'arcs $E_{\mathcal{F}}$ et E_y à partir de l'ensemble d'arcs $E_{\mathcal{G}}$ de telle sorte que :

1. pour tout couple de sommets i et j dans X il existe soit l'arc (i, j) soit l'arc (j, i) dans $E_x \cup E_y$;
2. les ensembles d'arcs $E_x \cup E_y$ et $E_x \cup (E_y)^-$ (les arcs de $(E_y)^-$ sont les arcs de (E_y) orientés dans le sens contraire) sont sans circuit.

Comme $E_{\mathcal{F}} \cup E_{\mathcal{G}}$ ne contient pas de circuit, on peut définir un ordre σ sur les sommets de X tel que s'il existe un arc (i, j) dans $E_{\mathcal{F}} \cup E_{\mathcal{G}}$ alors i est forcément avant j selon l'ordre σ. De même, comme $E_{\mathcal{F}} \cup E_{\mathcal{G}^-}$ ne contient pas de circuit, on peut définir un ordre τ sur les sommets de X tel que s'il existe un arc (i, j) dans $E_{\mathcal{F}} \cup (E_{\mathcal{G}^-})$ alors i est forcément avant j selon l'ordre τ. On construit alors E_x et E_y itérativement de la manière suivante. Initialement, on pose $E_x = E_{\mathcal{F}}$ et $E_y = E_{\mathcal{G}}$. Pour chaque couple de sommets i et j tel qu'il n'existe pas d'arcs (i, j) ni (j, i) dans $E_x \cup E_y$, on souhaite construire un nouvel arc, soit dans E_x, soit dans E_y. Pour cela, supposons que i est avant j dans σ, il y a alors deux cas possibles :

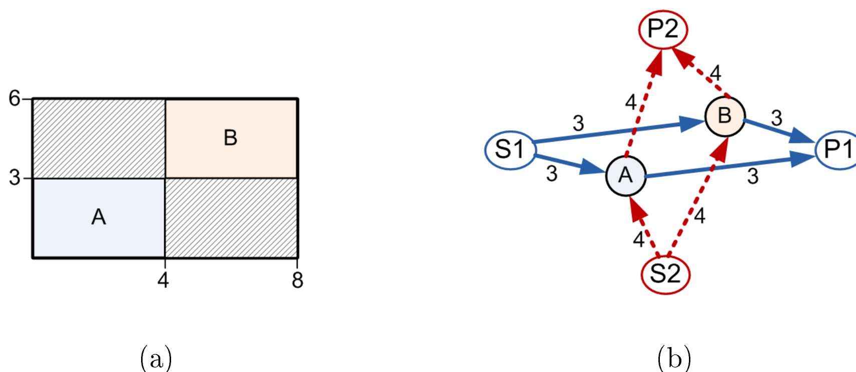
Premier cas : i est avant j dans τ. On crée alors l'arc (i, j) dans E_x ;

Second cas : i est après j dans τ. On crée alors l'arc (i, j) dans E_y.

Il est clair, qu'après l'ajout de l'un de ces deux arcs, les ordres σ et τ sont toujours respectés. On ne crée donc pas de circuit.

Notons que cette étape 1 est indispensable. En effet, tous les objets doivent être liés par un arc pour que leur position relative soit connue. Par exemple, considérons le placement, donné par la figure 2.22 (a), de deux objets A et B ainsi que le double flot déduit de ce placement, figure 2.22 (b).

La projection du côté gauche de B sur l'axe des ordonnées est disjointe de celle de A. Il en est de même pour la projection du côté bas de B sur l'axe des abscisses. On en déduit qu'il n'existe aucun arc portant du flot entre A et B. Donc il n'est pas possible de connaître la position relative des deux objets. De ce fait, un calcul direct des positions des objets (en utilisant uniquement les arcs portant du flot) donne $x_A = 0$, $y_A = 0$, $x_B = 0$, $y_B = 0$. On constate que les deux objets sont superposés. C'est pourquoi, il est nécessaire d'ajouter des arcs orientés qui respectent les listes σ et τ précalculées.



(a) (b)

FIG. 2.22 – Exemple de double flot pour deux objets qui ne partagent aucune abscisse ou ordonnée : (a) placement ; (b) double flot déduit

Etape 2

A tout arc (i, j) de E_x on associe le coût L_i (longueur de l'objet i). A tout arc (i, j) de E_y on associe le coût l_i (largeur de l'objet i). Les sommets fictifs source et puits sont supposés de longueur et largeur nulles. On peut alors calculer la position (x_i, y_i) de chaque objet i par :

- x_i est égal au plus long chemin de la source vers l'objet i dans le graphe G_x ;
- y_i est égal au plus long chemin de la source vers l'objet i dans le graphe G_y.

Etape 3

On vérifie que le placement défini par les positions calculées de cette manière respecte les dimensions du conteneur. Il suffit de montrer que pour tout sous-ensemble Y de X tel qu'il n'existe pas de chemin formé par les arcs $E_{\mathcal{G}}$ dans Y, on a $\sum_{y \in Y} L_y \leq L_C$ et que pour tout sous-ensemble Z de X tel qu'il n'existe pas de chemin formé par les arcs $E_{\mathcal{F}}$ dans Z, on a $\sum_{z \in Z} l_z \leq l_c$.

On montre la première inégalité, la deuxième se démontre de la même façon.

On procède par récurrence sur le nombre d'arcs dans $E_{\mathcal{G}}$. Il est clair que si $E_{\mathcal{G}} = \emptyset$, alors on a l'inégalité triviale $0 \leq 0$. Si $E_{\mathcal{G}}$ n'est pas vide, on choisit un sommet y dans Y et un chemin de $E_{\mathcal{G}}$ de source vers puits qui inclut y. On note δ la plus petite quantité de flot portée par un arc de ce chemin. Par définition, $\delta > 0$. D'après l'hypothèse de départ, le chemin ne contient aucun sommet de Y à part y. On peut alors remplacer la longueur totale L_C par L_C - δ et la longueur L_i des objets du chemin par L_i-δ. On enlève alors au flot $\mathcal{G}$ porté par les arcs du chemin la quantité δ. Il y a alors au moins un arc qui porte une quantité nulle de flot, il est donc supprimé de $E_{\mathcal{G}}$. On a ainsi réduit le nombre d'arcs dans $E_{\mathcal{G}}$. Par récurrence, on peut alors conclure que l'inégalité $\sum_{y \in Y} L_y \leq L_C$ est vérifiée.

Conclusion

Finalement, on a montré que s'il existe un double flot sans circuit associé à une instance du 2OPP, alors il est possible d'en déduire une solution du 2OPP. Comme on a montré précédemment qu'il est possible de construire un double flot sans circuit à partir d'une solution du 2OPP, on a montré l'équivalence entre l'existence d'une solution à une instance

de 2OPP et l'existence d'un double flot sans circuit associé à cette instance. □

2.6.2.5 Exemple de calcul d'une solution du 2OPP à partir d'un double flot sans circuit

La démonstration du théorème 5 donne une méthode pour obtenir un placement à partir d'un double flot sans circuit. On illustre ici cette méthode sur un exemple.

On utilise l'exemple donné au début de la section 2.6 pour illustrer le calcul d'une solution du 2OPP à partir d'un double flot sans circuit. La dimension des objets considérés est donnée tableau 2.16.

objets	1	2	3	4	5	6	7
longueur	3	2	3	4	2	2	2
largeur	3	3	2	2	1	4	2

TAB. 2.16 – Taille des objets

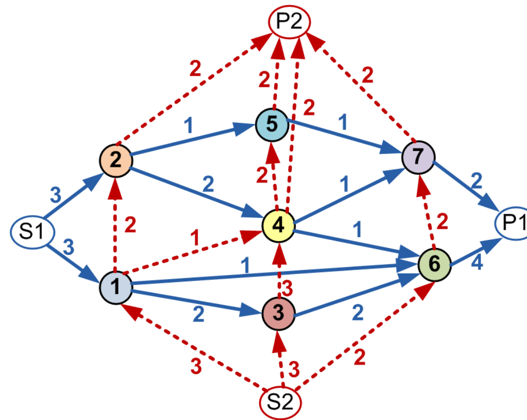


FIG. 2.23 – Double flot : les valeurs sur les arcs représentent la quantité de flot portée par l'arc

Le double flot considéré est donné figure 2.23. Dans cette figure, le flot $\mathcal{F}$ est porté par les arcs en traits pleins et le flot $\mathcal{G}$ est porté par les arcs en pointillés. A partir de ce double flot on détermine l'ensemble des arcs $E_\mathcal{F}$ et l'ensemble des arcs $E_\mathcal{G}$.

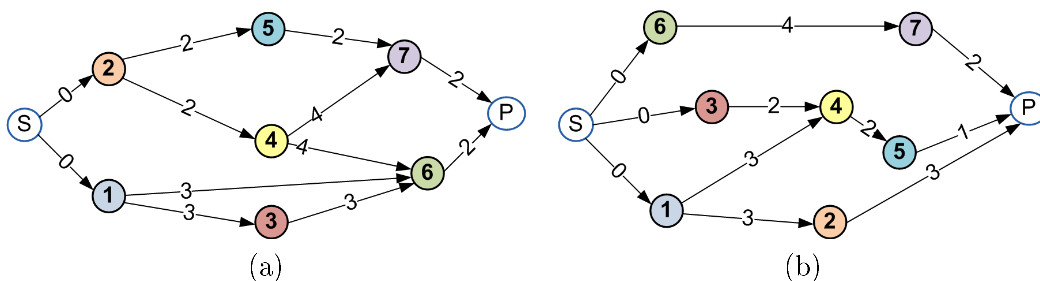


FIG. 2.24 – (a) : graphe $G_\mathcal{F}$; (b) : graphe $G_\mathcal{G}$. Les arcs sont valués par leur coût

La figure 2.24 représente les graphes $G_\mathcal{F} = (X^*, E_\mathcal{F})$ et $G_\mathcal{G} = (X^*, E_\mathcal{G})$. Rappelons qu'un arc (i, j) de l'ensemble $E_\mathcal{F}$ est valué par un coût égal à la longueur de l'objet i et

qu'un arc (i,j) de l'ensemble $E_{\mathcal{G}}$ est valué par un coût égal à la largeur de l'objet i.

Les seuls couples de sommets qui ne sont liés, ni par un arc de $E_{\mathcal{F}}$, ni par un arc de $E_{\mathcal{G}}$ (sans tenir compte de l'orientation) sont $(1,5)$, $(1,7)$, $(2,7)$, $(2,6)$, $(2,3)$, $(3,5)$, $(3,7)$ et $(5,6)$.

Afin de compléter les ensembles d'arcs $E_{\mathcal{F}}$ et $E_{\mathcal{G}}$ pour prendre en compte ces couples de sommets, il faut déterminer les listes σ et τ. Par exemple, on peut choisir les listes suivantes :

$\sigma = \{1,2,3,4,5,6,7\}$;

$\tau = \{2,5,4,1,3,7,6\}$.

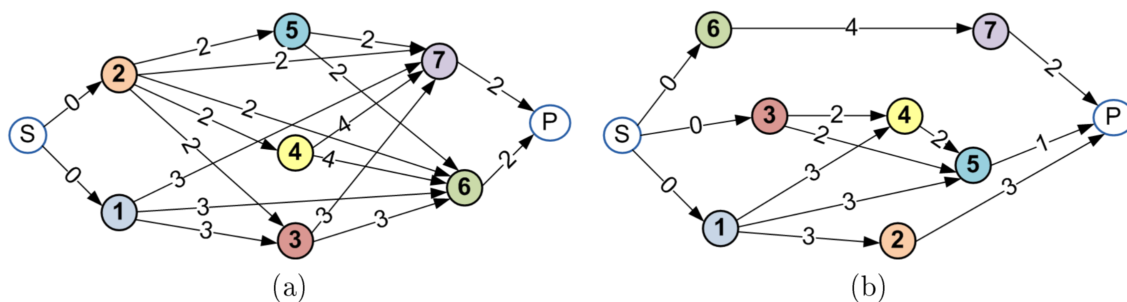


FIG. 2.25 – (a) : graphe G_x ; (b) : graphe G_y. Les arcs sont valués par leur coût

Il vient que $E_x = E_{\mathcal{F}} \cup \{(1,7),(2,7),(2,6),(2,3),(3,7),(5,6)\}$ et $E_y = E_{\mathcal{G}} \cup \{(1,5),(3,5)\}$. On construit alors les graphes $G_x = (X^*, E_x)$ et $G_y = (X^*, E_y)$ (voir figure 2.25).

Un calcul des plus longs chemins dans G_x^* donne $x_1 = 0$, $x_2 = 0$, $x_3 = 3$, $x_4 = 2$, $x_5 = 2$, $x_6 = 6$ et $x_7 = 6$. Un calcul des plus longs chemins dans G_y^* donne $y_1 = 0$, $y_2 = 3$, $y_3 = 0$, $y_4 = 3$, $y_5 = 4$, $y_6 = 0$ et $y_7 = 4$.

2.6.3 Proposition d'un algorithme pour le calcul d'un double flot sans circuit

L'algorithme proposé se déroule en deux temps. Dans un premier temps, le flot $\mathcal{F}$ est construit en relaxant le 2OPP en RCPSP. Dans un second temps, le flot $\mathcal{G}$ est construit de sorte qu'il forme un double flot sans circuit avec $\mathcal{F}$. On s'intéresse dans cette section uniquement à la construction du flot $\mathcal{G}$, les algorithmes pour le RCPSP ayant déjà été décrits dans les sections précédentes.

2.6.3.1 Principe général de construction de $\mathcal{G}$

Le principe général de construction du flot $\mathcal{G}$ est le suivant. A partir du flot $\mathcal{F}$ qui donne la position relative des objets de gauche à droite, on tente de créer des listes d'objets qui peuvent accepter du flot $\mathcal{G}$. Par exemple, si on crée la liste $\{a,b,c\}$ cela signifie que a reçoit une certaine quantité g de flot $\mathcal{G}$ de la source, a transmet ensuite ce flot à b, qui le transmet à c, qui le transmet au puits. La quantité de flot g dépend de la durée des actions.

Afin de visualiser ce processus, on donne un exemple sur un placement (figure 2.26). Il est clair qu'au moment où l'on construit les flots, on ne connaît pas encore ce placement,

il est obtenu seulement à la fin du processus. Mais cet exemple permet de visualiser les étapes qui ont été franchies lors de la construction du flot $\mathcal{G}$. La construction du flot $\mathcal{G}$ considère des tranches verticales dans le placement. Ces tranches doivent être le plus large possible et telles qu'aucun objet ne commence ou ne finisse au milieu d'une tranche. Pour le placement de la figure 2.26 on peut définir 5 tranches.

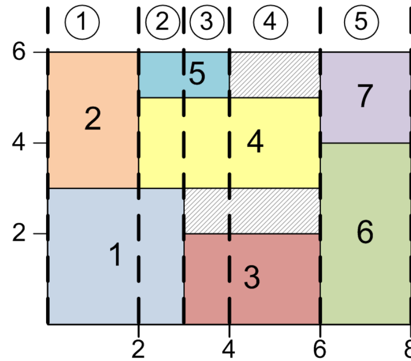


FIG. 2.26 – Tranches verticales dans un placement

On en déduit que l'algorithme a eu besoin de cinq itérations pour fixer complètement le flot $\mathcal{G}$:

itération 1 : on considère la tranche 1 de largeur 2. Elle définit la liste des objets $\{1, 2\}$. On fait passer $g = 2$ unités de flot $\mathcal{G}$ dans cette liste : $\mathcal{G}_{s,1} = 2$, $\mathcal{G}_{1,2} = 2$ et $\mathcal{G}_{2,p} = 2$;

itération 2 : on considère la tranche 2 de largeur 1. Elle définit la liste des objets $\{1, 4, 5\}$. On ajoute aux quantités de flots $\mathcal{G}$ éventuellement déjà présentes dans cette liste $g = 1$ unité de flot : on obtient $\mathcal{G}_{s,1} = 3$, $\mathcal{G}_{1,4} = 1$, $\mathcal{G}_{4,5} = 1$ et $\mathcal{G}_{5,p} = 1$;

⋮

itération 5 : on considère la tranche 5 de largeur 2. Elle définit la liste des objets $\{6, 7\}$. On ajoute aux quantités de flots $\mathcal{G}$ éventuellement déjà présentes dans cette liste $g = 2$ unités de flot $\mathcal{G}$ dans cette liste : on obtient $\mathcal{G}_{s,6} = 2$, $\mathcal{G}_{6,7} = 2$ et $\mathcal{G}_{7,p} = 2$.

Finalement, on obtient le flot $\mathcal{G}$ décrit par la figure 2.27.

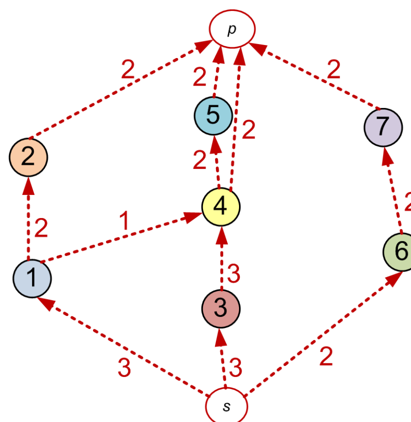


FIG. 2.27 – Flot $\mathcal{G}$ construit à partir des tranches définies par la figure 2.26

Remarque 7
L'ordre des objets dans les listes est très important. Par exemple, à l'itération 2, l'objet 2 a reçu suffisamment de flot et laisse donc sa place aux objets à sa droite : 4 et 5. Les objets 4 et 5 doivent donc prendre la place de 2 dans la liste. Si on considère la liste $\{4, 5, 1\}$, *par exemple, au lieu de* $\{1, 4, 5\}$ *on obtient une quantité de flot* $\mathcal{G}$ *non nulle sur l'arc (5,1). Comme on a déjà une quantité de flot* $\mathcal{G}$ *non nulle sur l'arc (1,2) et une quantité de flot* $\mathcal{F}$ *non nulle sur l'arc (2,5), on obtient le circuit (5,1,2,5) dans le graphe* $G_{\mathcal{F}\cup\mathcal{G}}$. *Ce double flot ne définit alors plus un placement.*

Remarque 8
Nous avons tout d'abord tenté une autre approche qui consiste à construire les deux flots $\mathcal{F}$ *et* $\mathcal{G}$ *simultanément. Pour cela on utilise des techniques de coupe et d'insertion comme nous l'avons fait pour le RCPSP (section 2.2). Cependant, il s'est avéré qu'il est assez lourd et fastidieux de construire ces flots simultanément à cause, entre autre, de la contrainte de « double flot sans circuit » qui doit être vérifiée à chaque itération. Les résultats numériques sont donc peu satisfaisants : les temps de calcul sont assez élevés et l'algorithme trouve peu souvent une solution (voir les résultats sur* `http://www.isima.fr/~toussain/` *qui complètent le rapport de recherche [QT10a]. Ce rapport décrit la méthode et les algorithmes).*

2.6.3.2 Description de l'algorithme

Initialement, les quantités de flot $\mathcal{G}$ sont nulles. La longueur totale de l'aire occupée par les objets est nulle. Elle va augmenter au fur et à mesure de la construction du flot $\mathcal{G}$. Lorsqu'un objet a reçu suffisamment de flot $\mathcal{G}$, il est marqué. L'algorithme s'arrête lorsque tous les objets sont marqués.

Le flot $\mathcal{G}$ est calculé tranche par tranche. Pour construire une tranche, il faut connaître la position relative des objets de gauche à droite : elle est donnée par le flot $\mathcal{F}$. Ainsi, on note $avant(i, j)$ si $\mathcal{F}_{ij} > 0$. Cela signifie que l'objet i doit être déjà marqué pour que l'objet j puisse recevoir du flot $\mathcal{G}$. Pour un objet j, on appelle degré de j, et on note $deg(j)$, le nombre de relations $avant(i, j)$ où i est un objet non encore marqué.

Une itération de l'algorithme de construction du flot $\mathcal{G}$ se déroule de la manière suivante :

1. Les sommets de degré 0 forment l'ensemble $Actif$ et sont dits « actifs » à l'itération courante ;
2. Les sommets actifs sont ordonnés de sorte qu'il est possible de mettre du flot $\mathcal{G}$ du premier objet vers le second, du second vers le troisième, etc, sans créer de circuit ;
3. Le degré des objets non marqués et non actifs peut éventuellement augmenter à cause de l'ordre choisi (ce phénomène est expliqué section 2.6.3.3) ;
4. On fait passer la quantité maximale de flot $\mathcal{G}$ du premier élément actif vers le dernier ;
5. Les objets qui ont reçu suffisamment de flot sont marqués et les degrés sont mis à jour en conséquence.

Cette procédure est donnée par l'algorithme 31.

Algorithme 31 : Construire_$\mathcal{G}$ (Construction du flot $\mathcal{G}$)

Entrées : X, $\mathcal{F}$

1 **pour chaque** $i \in X$ **faire**
2 $\quad deg(i)$ = nombre d'arcs (x, i) avec $x \in X$ qui portent du flot $\mathcal{F}$;
3 $\quad long(i)$ = longueur de l'objet i ;
4 $\quad avant(i, j) = 1$ si $\mathcal{F}_{ij} > 0$;
5 $\quad marq(i) = 0$;
6 $\quad longTotal = 0$;

7 **Tant que** *tous les jobs n'ont pas été marqués* **faire**
8 $\quad$ Actif = $\{i \in X$, marq(i) = 0 et deg(i) = 0 $\}$;
9 $\quad$ **Ordonnancer** l'ensemble Actif ;
10 $\quad$ **pour chaque** $x \in X$ *non marqué et non actif* **faire**
11 $\quad\quad$ **Si** $\exists i \in Actif, avant(i, x) = 1$ *et* $\exists j \in Actif, avant(j, x) = 1$ **alors**
12 $\quad\quad\quad$ **pour chaque** k *après* i *et avant* j *dans actif tel que* $avant(k, x) = 0$ **faire**
13 $\quad\quad\quad\quad avant(k, x) = 1$;
14 $\quad\quad\quad\quad deg(x) = deg(x) + 1$;

15 $\quad l_0 \leftarrow \min_{i \in Actif} long(i)$;
16 $\quad longTotal = longTotal + l_0$;
17 $\quad pred$ = source ;
18 $\quad$ **Pour** i = *premier élément de Actif à* i = *dernier élément de Actif* **faire**
19 $\quad\quad \mathcal{G}_{pred,i} \leftarrow \mathcal{G}_{pred,i} + l_0$;
20 $\quad\quad pred = i$;
21 $\quad\quad long(i) = long(i) - l_0$;
22 $\quad\quad$ **Si** $long(i) = 0$ **alors**
23 $\quad\quad\quad$ Supprimer i de Actif ;
24 $\quad\quad\quad$ Marquer i : $marq(i) = 1$;
25 $\quad\quad\quad$ Décrémenter de 1 les $x \in X$ tels qu'il existe un arc de i vers x ;

26 $\quad \mathcal{G}_{i,puits} \leftarrow \mathcal{G}_{i,puits} + l_0$;

2.6.3.3 Un point délicat de l'algorithme : ordonner les objets actifs

Dans l'algorithme de construction de $\mathcal{G}$, le degré d'un objet i détermine le nombre d'objets qui doivent déjà être marqués pour que i puisse devenir actif. Cette quantité peut augmenter au fur et à mesure de l'algorithme en fonction des choix effectués pour ordonner l'ensemble des objets actifs. Pour illustrer ce phénomène, considérons l'exemple à quatre objets de la figure 2.28.

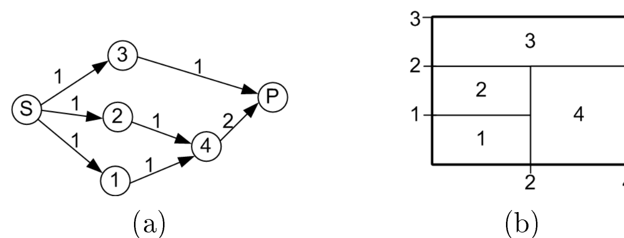


FIG. 2.28 – (a) Flot $\mathcal{F}$; (b) Placement optimal pour les positions relatives données par $\mathcal{F}$

Dans la figure 2.28, les degrés initiaux des sommets sont $deg(1) = 0$, $deg(2) = 0$, $deg(3) = 0$, et $deg(4) = 2$. A la première itération, l'ensemble des jobs actifs est donc formé des objets 1, 2 et 3. Il y a donc 3! possibilités pour ordonner ces objets. Si on choisit l'ordre 1, 2 et 3, alors les objets sont positionnés comme indiqué par la figure 2.29(a). Il est clair qu'à l'itération suivante l'objet 4, devient actif et sera donc positionné à l'abscisse 2 (qui donne le placement optimal). Par contre, si on choisit l'ordre initial 1, 3 et 2, alors les objets sont positionnés comme indiqué par la figure 2.29(b). Il est clair que l'objet 4 ne peut plus être placé juste derrière 1 et 2. Il devra être placé après l'objet 3, ce qui a pour conséquence de placer l'objet 4 à l'abscisse 4. On ajoute alors la relation $avant(3, 4)$ et on augmente le degré de 4.

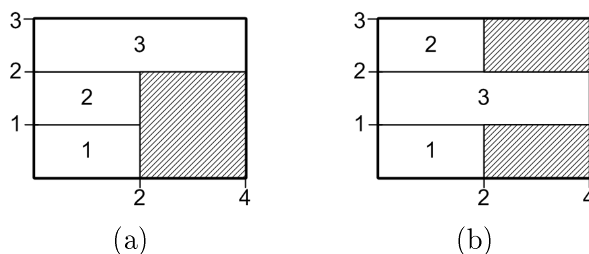


FIG. 2.29 – (a) : placement des trois premiers objets avec $Actif = \{1, 2, 3\}$; (b) : placement des trois premiers objets avec $Actif = \{1, 3, 2\}$

On déduit de cet exemple que l'ordre dans lequel on ordonne les objets actifs est très important. Il faut éviter d'augmenter le nombre de relations *avant* car cela a pour effet de dégrader le placement. Dans ce but, les objets qui donnent du flot $\mathcal{F}$ à un même objet doivent être, autant que possible, consécutifs dans $Actif$.

2.6.3.4 L'algorithme complet de construction du double flot pour la résolution du 2OPP

L'algorithme SGS de la section précédente pour la résolution du RCPSP est adapté pour calculer le flot $\mathcal{F}$ au fur et à mesure des itérations. On appelle `Construire_`$\mathcal{F}$ l'adaptation de cet algorithme. Le principe est ensuite le même que pour le *double SGS* :

- dans un premier temps, on résout le problème de RCPSP associé au 2OPP. La solution est définie par le flot $\mathcal{F}$ et doit avoir un makespan inférieur ou égal à la longueur du conteneur L_C. Si le makespan obtenu est supérieur à L_C, alors on recommence cette étape (un nombre limité de fois) pour essayer de trouver une meilleure solution.
- dans un second temps, si une solution de makespan inférieur ou égal à L_C a été trouvée, on construit le flot $\mathcal{G}$. Si le flot $\mathcal{G}$ calculé est compatible avec la longueur L_C, alors le double flot $(\mathcal{F},\mathcal{G})$ représente une solution du 2OPP. Sinon, on recommence cette étape un nombre limité de fois.

La construction du flot $\mathcal{G}$ dépend beaucoup du flot $\mathcal{F}$ puisque c'est à partir de ce flot que sont calculées les tranches. C'est pourquoi, il est avantageux de recommencer plusieurs fois le schéma de construction de $\mathcal{F}$, puis de $\mathcal{G}$. L'algorithme complet du « double flot » est donné par l'algorithme 32.

Algorithme 32 : Double_Flot

```
   Entrées : B, itMax1, itMax2, maxRepet, L_C, l_C
   Sorties : F, G
1  Déduire de l'ensemble des objets B l'ensemble X des activités du RCPSP ;
2  res ← faux ; k ← 0 ;
3  Tant que res = faux et k < maxRepet faire
4  |  i ← 0 ;
5  |  Tant que res = faux et i < itMax1 faire
6  |  |  res ← Construire_F(X, L_C) ;
7  |  |  i ← i + 1 ;
8  |  Si res = vrai alors
9  |  |  res ← faux ;
10 |  |  i ← 0 ;
11 |  |  Tant que res = faux et i < itMax2 faire
12 |  |  |  res ← Construire_G(X, F) ;
13 |  |  |  i ← i + 1 ;
```

2.6.3.5 Principales différences entre le double SGS et le double flot

Les deux méthodes sont basées sur la même idée : la relaxation du problème de 2OPP en RCPSP qui implique une résolution en deux phases :

1. résolution du RCPSP ;
2. résolution du 2OPP à partir de la solution du RCPSP.

Cependant, il existe diverses manières de tirer profit de ce schéma.

Le double SGS calcule directement les positions : la phase 1 donne l'abscisse des objets, la phase 2 donne l'ordonnée des objets. Le défaut de ce schéma est qu'il manque de souplesse : au terme de la phase 1 les abscisses sont fixées donc, si le makespan C_{max} trouvé est strictement inférieur à L_C, alors durant la phase 2 on tente de construire un placement pour un conteneur de longueur C_{max}. Ceci est désavantageux car il se peut qu'il soit impossible de trouver un placement dans ce conteneur réduit alors qu'il en existe un dans le conteneur initial. La qualité de ce schéma est qu'il est très simple et très rapide, ce qui permet de l'appliquer un grand nombre de fois et donc d'augmenter les chances de trouver une solution.

Le double flot calcule uniquement des positions relatives : la phase 1 donne la position relative des objets de gauche à droite et la phase 2 donne les positions relatives de bas en haut. L'intérêt de ce schéma, par rapport au précédent, réside dans le fait que les abscisses ne sont pas fixées au terme de la phase 1. L'inconvénient est que le flot $\mathcal{F}$ contraint beaucoup la construction de $\mathcal{G}$: à chaque itération, $\mathcal{F}$ détermine non seulement les objets actifs mais aussi, en partie, l'ordre de ces objets dans l'ensemble $Actif$ (à partir de la deuxième itération). De ce fait, la position en ordonnée des objets est contrainte par $\mathcal{F}$.

2.6.4 Expérimentations numériques

Nous comparons le double flot avec la méthode *double SGS* proposée dans la section précédente. Nous utilisons les mêmes instances et les conditions expérimentales sont identiques (C++, PC AMD Opteron, 2.1GHz sous Linux).

On donne également dans ce tableau les résultats obtenus par les méthodes exactes de la littérature :
- OPP de Fekete *et al.* ([FSdV07]) ;
- TSBP ([CCM07]) et ER ([CJCM08]) de Clautiaux *et al.* ;
- SWEEP de Beldiceanu et Carlsson ([BC01]).

Les résultats sont synthétisés dans le tableau 2.17. La colonne « Instances » donne le nom de l'instance, le nombre d'objets à placer et la faisabilité de l'instance. La colonne « Méthodes exactes » donne le temps cpu en secondes mis par la méthode concernée pour déterminer si l'instance est faisable ou pas. Le symbole « - » signifie que la méthode n'a pas trouvée de réponse dans le temps qui lui était imparti (15 minutes pour OPP et SWEEP, 4 minutes pour TSBP).

Le double flot est exécuté avec les paramètres suivants : $itMax1 = 10\ 000$ $itMax2 = 100$ et $maxRepet = 10$. Il est exécuté cinq fois. La colonne correspondante est divisée en trois sous-colonnes :
- la colonne « réal. » contient *non*, si aucune solution n'a été trouvée. Elle contient *oui*, si une solution a été trouvée au moins une fois. Dans ce cas, on indique sur les 5 exécutions combien ont mené à une solution. Le symbole « * » signifie que l'heuristique fournit la bonne réponse ;
- la colonne « cpu moy. » donne le temps cpu moyen en secondes d'une exécution ;
- la colonne « best cpu » donne le meilleur temps cpu sur les cinq exécutions.

Les mêmes informations sont indiquées pour le *double SGS*.

On constate que le double flot parvient à trouver 13 solutions sur 15 instances faisables. Le *double SGS* en trouve également 13. Les deux instances qui posent problème ne sont pas les mêmes pour le double flot et le *double SGS*. Seule l'instance E02F17 n'est résolue par aucune des deux méthodes. Les temps de calcul du double flot sont environ 40 % plus élevés que ceux du double SGS mais ils restent compétitifs vis à vis des méthodes de la littérature.

Instances			Méthodes exactes				Heuristiques							
			OPP	TSBP	SWEEP	ER	double SGS				double flot			
nom	nb objets	réalisable	[FSdV07]	[CCM07]	[BC01]	[CJCM08]	cette thèse							
			cpu	cpu	cpu	cpu	réal.		cpu moy.	best cpu	réal.		cpu moy.	best cpu
E00N23	23	non	-	289	-	6.75	non	*	0.72	0.71	non	*	0.92	0.89
E00X23	23	non	-	86	-	1.2	non	*	0.71	0.70	non	*	0.89	0.89
E05N15	15	non	0	0	-	0.01	non	*	0.42	0.42	non	*	0.50	0.49
E05X15	15	non	2	0	17.03	0.2	non	*	0.42	0.41	non	*	0.50	0.49
E05F20	20	oui	491	2	0	1.63	oui(5/5)	*	0.04	0.00	oui(5/5)	*	0.01	0.00
E04F20	20	oui	22	3	11.96	1.4	oui(5/5)	*	0.01	0.00	oui(5/5)	*	0.01	0.00
E13N15	15	non	0	0	0.08	0.01	non	*	0.42	0.41	non	*	0.49	0.48
E10N15	15	non	0	0	-	0.06	non	*	0.44	0.43	non	*	0.51	0.50
E03N16	16	non	2	32	62.9	1.02	non	*	0.47	0.46	non	*	0.56	0.55
E20F15	15	oui	0	1	0.01	0.6	oui(5/5)	*	0.00	0.00	oui(5/5)	*	0.00	0.00
E04N15	15	non	0	1	11.04	0.79	non	*	0.43	0.43	non	*	0.51	0.50
E03N15	15	non	0	1	1.16	0.72	non	*	0.43	0.43	non	*	0.50	0.50
E10X15	15	non	0	1	6.85	0.46	non	*	0.43	0.43	non	*	0.51	0.51
E07N15	15	non	0	1	10.39	0.59	non	*	0.42	0.41	non	*	0.50	0.49
E05N17	17	non	0	1	11.23	0.37	non	*	0.49	0.48	non	*	0.59	0.58
E08N15	15	non	0	1	2.11	0.33	non	*	0.42	0.41	non	*	0.49	0.48
E07F15	15	oui	0	1	1.88	0.35	oui(5/5)	*	0.00	0.00	oui(5/5)	*	0.00	0.00
E03X18	18	oui	0	22	0.33	0.55	oui(5/5)	*	0.15	0.02	oui(1/5)	*	3.58	0.53
E04N18	18	non	10	7	329.21	0.31	non	*	0.51	0.50	non	*	0.62	0.61
E02F20	20	oui	-	12	42.25	0.84	oui(3/5)	*	0.46	0.10	oui(2/5)	*	1.81	0.37
E04F17	17	oui	13	26	0	0.54	oui(5/5)	*	0.04	0.00	oui(5/5)	*	0.03	0.00
E13N10	10	non	0	0	0.51	0.16	non	*	0.26	0.25	non	*	0.28	0.28
E04F15	15	oui	0	1	1.29	0.43	oui(3/5)	*	0.35	0.18	non		2.12	0.50
E03N17	17	non	0	4	0.89	0.4	non	*	0.47	0.46	non	*	0.58	0.57
E00N15	15	non	0	2	136.59	0.44	non	*	0.43	0.42	non	*	0.50	0.50
E20X15	15	oui	0	44	0	0.24	non		0.00	0.00	oui(5/5)	*	0.00	0.00
E05F15	15	oui	0	3	0.27	0.3	oui(5/5)	*	0.01	0.00	oui(5/5)	*	0.06	0.00
E02F17	17	oui	7	12	0.97	0.52	non		0.51	0.50	non		0.60	0.60
E02F22	22	oui	167	4	0.38	0.79	oui(5/5)	*	0.03	0.00	oui(5/5)	*	0.05	0.01
E04F19	19	oui	560	7	0.08	0.77	oui(5/5)	*	0.01	0.00	oui(5/5)	*	0.02	0.00
E05F18	18	oui	0	126	0.02	0.51	oui(5/5)	*	0.06	0.01	oui(5/5)	*	0.15	0.02
E08F15	15	oui	0	117	0	0.38	oui(5/5)	*	0.01	0.00	oui(5/5)	*	0.08	0.00
E04N17	17	non	0	1	2.31	0.23	non	*	0.48	0.46	non	*	0.57	0.57
E13X15	15	non	0	0	9.34	0.07	non	*	0.40	0.39	non	*	0.47	0.47
E15N15	15	non	0	0	6.78	0.17	non	*	0.42	0.41	non	*	0.49	0.49
E00N10	10	non	0	0	0.04	0.01	non	*	0.28	0.27	non	*	0.30	0.29
E02N20	20	non	0	1	2	0.32	non	*	0.58	0.57	non	*	0.72	0.72
E03N10	10	non	0	0	0	0.09	non	*	0.27	0.27	non	*	0.29	0.29
E07N10	10	non	0	0	0.19	0.17	non	*	0.27	0.27	non	*	0.29	0.29
E07X15	15	non	0	0	0.06	0	non	*	0.42	0.41	non	*	0.49	0.48
E10N10	10.00	non	0	0	0.02	0.05	non	*	0.27	0.27	non	*	0.30	0.29
moyenne			33.53	19.73	18.11	0.60			0.32	0.29			0.53	0.37

TAB. 2.17 – Comparaison des résultats de cette thèse sur le 2OPP avec ceux de la littérature (tableau extrait de [CJCM08] et mis à jour avec nos méthodes)

2.6.5 Conclusion sur le double flot

Cette section présente une modélisation à base de flots pour un problème de placement. Cette approche est originale pour deux raisons :
- à notre connaissance, il s'agit de la première méthode pour résoudre un problème de placement à l'aide des flots ;
- l'approche utilisée est basée sur la relaxation du problème de placement en un RCPSP, ce qui constitue également une approche nouvelle.

Nous avons montré l'équivalence entre une solution d'un problème de placement orthogonal en deux dimensions (2OPP) et un multiflot qui possède des caractéristiques particulières. Ce multiflot est composé de deux flots : le premier donne la position relative des objets de gauche à droite, le second donne la position relative des objets de bas en haut. L'union de ces deux flots, que l'on nomme double flot, permet de positionner exactement les objets.

L'algorithme proposé a été testé et comparé à quatre méthodes de la littérature. Les résultats montrent que le double flot résout efficacement le problème de placement : bien que la méthode soit heuristique, elle trouve presque toujours la solution et les temps de calcul sont meilleurs que ceux des méthodes exactes. Le double flot a également été comparé à une autre méthode basée sur la relaxation du 2OPP en RCPSP que nous avons développée. Le nombre de solutions trouvées est similaire pour ces deux méthodes, les temps de calcul sont un peu plus élevés pour le double flot.

2.7 Conclusion

Dans ce chapitre, nous nous sommes intéressé à un problème d'ordonnancement de projet sous contraintes de ressources : le *Resource-Constrained Project Scheduling Problem* (RCPSP). Le RCPSP est un problème classique dans lequel on doit planifier des activités qui ont besoin de ressources au moment de leur exécution. Des contraintes de précédence peuvent également exister. L'objectif est de trouver un ordonnancement des activités qui minimise la durée totale du projet et tel que, à tout instant, les consommations respectent la disponibilité des ressources. Notre objectif était de concevoir un modèle facilement adaptable à des extensions. Nous avons élaboré une méthode de résolution basée sur les flots. Cette méthode a été testée et comparée aux méthodes de la littérature. Bien que cette méthode n'ait pas été conçue dans le but de fournir les meilleurs résultats pour le RCPSP, elle se situe parmi les meilleures méthodes.

De plus, la modélisation et les algorithmes que nous avons obtenus pour ce problème nous ont permis de considérer :
- des extensions originales du RCPSP qui incluent des échanges supplémentaires entre les ressources ;
- un problème de placement dont la relaxation se ramène à un RCPSP.

Ainsi, nous avons, tout d'abord, traité une extension dans laquelle des contraintes temporelles particulières viennent s'ajouter aux contraintes de précédence. Ces contraintes représentent des temps de transport concernant l'acheminement des ressources entre les activités. La modélisation et les algorithmes de flots que nous avons conçus pour le RCPSP s'adaptent particulièrement bien à cette extension. Nous avons également envisagé le cas avec flux financiers. Ensuite, nous avons considéré un problème de placement orthogonal

en deux dimensions : le *Two Orthogonal Packing Problem* (2OPP). Nous avons proposé pour ce problème deux méthodes originales de résolution. Ces méthodes sont basées sur la relaxation du 2OPP en RCPSP ce qui constitue, à notre connaissance, une nouvelle approche. La première méthode utilise un schéma de génération d'ordonnancements similaire aux schémas existants pour le RCPSP. La deuxième approche utilise un multiflot. Nous avons montré l'équivalence entre l'existence d'un multiflot possédant certaines propriétés et une solution du 2OPP. Pour finir, nous avons testé ces deux méthodes sur les instances de la littérature et constaté qu'elles sont plus rapides que les méthodes de la littérature et fournissent souvent des résultats exacts bien que nos schéma d'optimisation soient heuristiques.

Chapitre 3

Deux problèmes de Ramassage et Livraison : le *Stacker Crane* et le *Dial-a-Ride*

Ce chapitre concerne les problèmes de tournées de type ramassage et livraison connus sous l'acronyme PDP pour *Pickup and Delivery* en anglais. Il s'agit de problèmes de tournées particuliers dans lesquels les demandes de transport émanent d'un nœud du réseau à destination d'un (ou plusieurs) autres nœuds du réseau. Parmi ces problèmes nous étudions : le *Stacker Crane Problem* et le *Dial-a-Ride Problem*. Dans un premier temps, nous donnons une présentation des différents problèmes de ramassage et livraison, ce qui permet de situer les deux problèmes étudiés. Ensuite, nous proposons une approche originale pour résoudre le *Stacker Crane Problem* dans sa forme préemptive. Enfin, nous abordons une nouvelle extension pour le *Dial-a-Ride Problem*.

3.1 Présentation générale des problèmes de Ramassage et Livraison (*Pickup and Delivery problems*)

Les problèmes de ramassage et livraison, plus connus sous le terme anglais *Pickup and Delivery Problems* (et notés PDP), constituent une classe de problèmes très variés. Dans un problème de *Pickup and Delivery*, on cherche à transporter des biens ou des personnes depuis une origine vers une destination grâce à une flotte de véhicules entreposés à un dépôt. Concrètement, ce type de problème peut s'appliquer à des transports scolaires, distribution de colis, transports pour personnes âgées ou handicapées, mouvements des robots sur une chaîne de production, ...

On représente habituellement ces problèmes à l'aide d'un graphe $G = (X, E)$ complet et orienté. On désigne par $X = \{0 \dots n\}$ l'ensemble des nœuds du graphe, le nœud 0 représentant le dépôt, les autres nœuds étant généralement appelés clients (on verra que dans la plupart des problèmes c'est effectivement ce qu'ils représentent). On désigne par E l'ensemble des arcs du graphe, ils sont valués par des coûts positifs. On dispose d'une flotte F de m véhicules, pour traiter un ensemble D de demandes. On définit une demande par son origine, sa destination et sa charge (quantité de marchandises à transporter, une marchandise désigne un bien ou une personne). On dit qu'une demande est satisfaite si sa

charge est transportée de son origine vers sa destination. Chaque véhicule dispose d'une capacité Q correspondant au nombre de marchandises qu'il peut transporter. Un nœud du graphe, autre que le dépôt, peut donc représenter :
- l'origine d'une demande,
- la destination d'une demande,
- un relais (nœud permettant de déposer temporairement la charge associée à une demande pour revenir la chercher plus tard, éventuellement à l'aide d'un second véhicule)

On désigne par *tournée* un chemin dans le graphe G qui commence et finit au dépôt. On appelle *coût d'une tournée* la somme des coûts des arcs empruntés par la tournée (en pratique cela correspond, en général, au nombre de kilomètres parcourus par le véhicule réalisant la tournée). Le but est de construire au plus m tournées de manière à satisfaire toutes les demandes en respectant les contraintes de capacité des véhicules et en minimisant le coût total des tournées. Notons que d'autres contraintes peuvent être ajoutées telles que des dates de passage chez les clients (ces dates se traduisent souvent par des fenêtres de temps comme nous le verrons dans la section 3.3). D'autres critères visant à prendre en compte la qualité du service sont parfois envisagés tels que la durée maximale du trajet pour une marchandise.

Nous introduisons ici les différents problèmes de *Pickup and Delivery.* On s'intéresse uniquement à des problèmes statiques (toutes les données sont connues a priori). Dans [BCGL07] Berbeglia *et al.* propose une classification détaillée des problèmes de *Pickup and Delivery* ainsi qu'une étude des méthodes utilisées. Une autre classification est proposée dans [PDH08a] et [PDH08b] : ces deux articles complémentaires présentent également les problèmes et les méthodes de résolution (exactes et heuristiques).

Dans [BCGL07] Berbeglia *et al.* proposent de représenter les problèmes de *Pickup and Delivery* suivant le schéma [Structure|Visite|Véhicule] comprenant trois champs :

1. Le champ « Structure » précise le nombre d'origines et destinations. On distingue trois cas :
   - 1-1 : chaque demande a une origine et une destination données (*one-to-one problems*) ;
   - M-M : n'importe quel nœud du graphe peut servir d'origine ou de destination pour n'importe quelle demande (*many-to-many problems*) ;
   - 1-M-1 : les échanges s'effectuent uniquement entre dépôt et clients : le dépôt est soit l'origine d'une demande soit la destination. (*one-to-many-to-one problems*).
2. Le champ « Visite » précise la manière dont on visite les nœuds. On distingue trois cas :
   - PD : chaque nœud est visité une unique fois pour être livré et collecté ;
   - P-D : chaque nœud peut être livré et collecté en même temps ou séparément ;
   - P/D : chaque nœud est soit livré soit collecté mais pas les deux.

   De plus il est habituel d'utiliser la lettre T s'il existe des nœuds qui peuvent servir de relais.
3. Le champ « Véhicule » donne le nombre de véhicules utilisés.

Un champ non spécifié est noté « - ».

3.1.1 Problème avec une origine et une destination données pour chaque demande *one-to-one problems*[1-1|-|-]

Dans les problèmes « un-à-un » un nœud du graphe G autre que le dépôt est origine ou destination d'une unique demande. On peut citer quatre problèmes connus de ce type : le *Vehicle Routing Problem with Pickups and Deliveries* [1-1|P/D|-], le *Dial-a-Ride Problem* [1-1|P/D|-], le *Traveling Salesman Problem with Pickup and Delivery* [1-1|P/D|1] et le *Stacker Crane Problem* [1-1|P/D|1].

***Vehicle Routing Problem with Pickups and Deliveries* (VRPPD)** [1-1|P/D|-]
Dans ce problème, on prend en compte un ensemble de demandes ayant chacune une charge, une origine et une destination données. On dispose d'une flotte de véhicules pour transporter toutes les marchandises depuis leur origine vers leur destination. On cherche les tournées qui minimisent le kilométrage total et satisfont toutes les demandes.

***Dial-a-Ride Problem* (DARP)** [1-1|P/D|-]
Ce problème représente un ensemble de demandes de transports faites à un opérateur. Le DARP et le VRPPD se définissent de la même façon mais le premier est associé à du transport de biens, tandis que le second est associé à un transport de personnes. Dans le DARP, la fonction objectif prend en compte la qualité de service, c'est pourquoi il est particulièrement bien adapté à la modélisation de transport de passagers. Une demande possède une origine donnée et une destination donnée. Elle inclut une heure de départ et/ou une heure d'arrivée approximative (qui se modélise souvent par une fenêtre de temps) et un nombre de passagers à transporter. On dispose de plusieurs véhicules pour traiter les demandes. Le but est de trouver des tournées qui satisfont toutes les demandes en minimisant les coûts (kilométrage total) et en optimisant la qualité de service (minimisation des temps d'attente et/ou des temps de transport pour les clients par exemple). Un exemple classique est le transport de personnes à mobilité réduite.

Nous reviendrons en détail sur ce problème dans la section 3.3.

***Traveling salesman problem with pickup and delivery* (TSPPD)** [1-1|P/D|1]
Ce problème est la version à un véhicule du VRPPD.

***Stacker Crane Problem* (SCP)** [1-1|P/D|1]
Ce problème est la version du TSPPD dans laquelle le véhicule est de capacité unitaire. En effet, dans ce problème, un véhicule doit satisfaire un ensemble de demandes possédant chacune une origine, une destination donnée et une charge unitaire à transporter de l'origine vers la destination. Il s'agit donc de trouver une tournée, la plus courte possible, qui commence et finit au dépôt et qui satisfait toutes les demandes. Le *stacker crane problem* est inspiré du problème de chargement / déchargement des navires à l'aide de grues-portiques. Ces grues ne peuvent manipuler qu'un conteneur à la fois. Il faut donc porter une attention particulière à l'ordre des chargements et des déchargements pour minimiser les déplacements de la grue.

Nous reviendrons en détail sur ce problème dans la section 3.2.

3.1.2 Problèmes avec plusieurs origines et plusieurs destinations (*Many-to-many problems*) [M-M|-|-]

Dans les problèmes « plusieurs-à-plusieurs » un nœud du graphe G peut être origine ou destination de n'importe quelle demande. On peut citer trois problèmes de ce type : le *Swapping Problem* [M-M|P-D|1], le *One-Commodity Pickup and Delivery Traveling Salesman Problem* [M-M|P/D|1] et le *Q-Delivery Traveling Salesman Problem*[M-M|P/D|1]. La différence avec les problèmes précédents réside dans le fait qu'une origine n'est pas associée à une unique destination mais peut être collectée pour livrer plusieurs destinations. En effet, les nœuds origine sont collectés (nœuds de collecte) pour livrer les nœuds destination (nœuds de livraison). Ainsi, une origine qui possède une marchandise de type A peut être collectée pour livrer n'importe quelle destination qui réclame une marchandise de type A.

***Swapping Problem* (SP)** [M-M|P-D|1]
Dans ce problème, on considère un unique véhicule de capacité unitaire. Il échange des marchandises entre les sommets. En chaque sommet, on collecte un type de marchandises et on livre un type de marchandises différent. Le nombre de marchandises à collecter est égal au nombre de marchandises à livrer dans chaque type, de telle sorte que le problème consiste en un réarrangement des marchandises entre les sommets. Le but est de trouver une tournée de coût minimal qui échange correctement les marchandises. Dans le cas préemptif, ce problème se nomme *Mixed Swapping Problem*.

***One-Commodity Pickup and Delivery Traveling Salesman Problem* (1-PDTSP)** [M-M|P/D|1]
Il s'agit d'une généralisation du célèbre voyageur de commerce (*Traveling Salesman Problem* - TSP). L'ensemble des nœuds est partagé entre nœuds de collecte et nœuds de livraison. Toutes les marchandises sont du même type, on peut livrer un nœud de livraison avec n'importe quelle marchandise collectée à un nœud de collecte.

***Q-Delivery Traveling Salesman Problem* (QDTSP)** [M-M|P/D|1]
Il s'agit d'un cas particulier du 1-PDTSP dans lequel on ne traite que des charges unitaires.

3.1.3 Problème du type *one-to-many-to-one pickup and delivery problems* [1-M-1|-|-]

Ce type de problème se rencontre par exemple dans le cadre de services où des clients se font livrer des conteneurs pleins et/ou rendent des conteneurs vides. On distinguera deux catégories : le 1-M-1-PDP avec demandes combinées et le 1-M-1-PDP avec demandes simples.

le 1-M-1-PDP avec demandes combinées [1-M-1|P-D|-]
Les nœuds sont à la fois nœuds de livraison et nœuds de collecte (les clients se font livrer **et** rendent).

le 1-M-1-PDP avec demandes simples [1-M-1|P/D|-]
Les nœuds sont soit nœuds de livraison soit nœuds de collecte (les clients se font livrer **ou** rendent mais pas les deux).

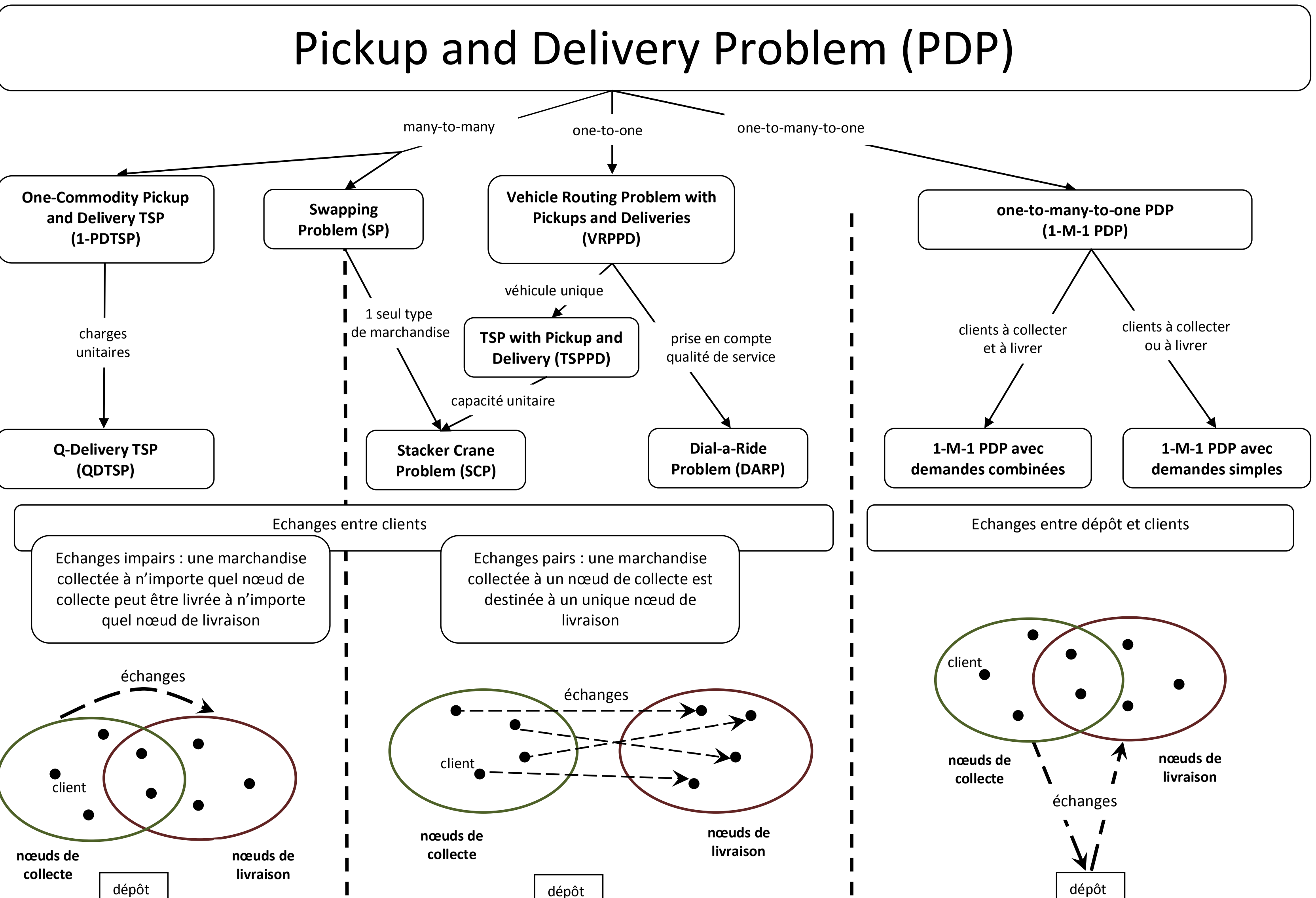


Fig. 3.1 – Synthèse des différents problèmes classiques de *Pickup and Delivery*

Le schéma de la figure 3.1 synthétise les relations entre les différents problèmes de *Pickup and Delivery* présentés dans cette section.

Dans le cadre de cette thèse, nous nous sommes intéressé plus particulièrement à la résolution heuristique du *Stacker Crane Problem* (SCP) sous sa forme préemptive et asymétrique ainsi qu'à la résolution heuristique du *Dial-a-Ride Problem* (DARP) dans lequel nous considérons une contrainte financière supplémentaire.

3.2 Le *Stacker Crane Problem* Préemptif et Asymétrique

Cette section s'intéresse au *Stacker Crane Problem* Préemptif et Asymétrique (SCPPA). Comme on l'a vu dans la section 3.1, le *Stacker Crane Problem* (SCP) est un problème de *pickup and delivery* caractérisé par le fait qu'on dispose d'un unique véhicule, ne transportant qu'une demande à la fois. Ce problème tire son nom des grues-portiques utilisées notamment pour le chargement et le déchargement des navires.

Le *stacker crane problem* peut se définir de la manière suivante : étant donné un graphe G dont les arcs sont valués par des longueurs (ou coûts) et un ensemble de demandes K, il s'agit de déterminer la tournée d'un unique véhicule V de telle sorte que chaque demande $k \in K$ soit transportée de son nœud origine o_k à son nœud destination d_k. L'ensemble des nœuds $X = \{0, 1, \ldots, n\}$ est tel que le nœud 0 représente le dépôt, les autres nœuds sont : soit l'origine d'une demande, soit sa destination. Chaque demande k est associée à une charge unitaire et le véhicule V ne peut transporter plus d'une charge à la fois. Il s'agit donc de trouver une tournée dans G, la plus courte possible, qui commence et finit au nœud 0 (dépôt) et qui permet à V de satisfaire toutes les demandes. Dans le *stacker crane problem* Préemptif, chaque charge peut être déchargée en n'importe quel nœud du graphe avant d'être rechargée. Ces déchargements / rechargements n'induisent pas de coût supplémentaire. Notons que dans les problèmes de *pickup and delivery* avec transbordement les demandes peuvent être également acheminées en plusieurs fois, mais les véhicules déchargeant et rechargeant une demande sont différents, ce terme est donc inapproprié ici. On parle de *stacker crane problem* asymétrique lorsque le coût d'un arc (x, y) peut être différent du coût de l'arc (y, x).

A notre connaissance, la version préemptive du SCP a été très peu étudiée. L'intérêt d'introduire des relais (préemption) est de réduire le coût de la tournée. La figure 3.2 (a) montre le trajet d'un véhicule qui part du dépôt puis traite dans l'ordre les demandes 1, 2 et 3. La figure 3.2 (b) montre l'intérêt d'ajouter un relais dans la tournée de la figure 3.2 (a) : le véhicule part alors du dépôt puis traite la demande 1. Il va ensuite à l'origine de la demande 2 mais ne la traite pas immédiatement, il pose sa charge au nœud relais et va traiter la demande 3. Enfin, il revient au nœud relais pour récupérer la charge de la demande 2 et l'emmène à destination. Nous avons choisi d'utiliser des distances euclidiennes dans les figures.

3.2.1 Etat de l'art

Le *stacker crane problem* a tout d'abord été introduit par Frederickson *et al.* dans [FHK78], sous sa forme symétrique et non préemptive. Ces auteurs fournissent une preuve

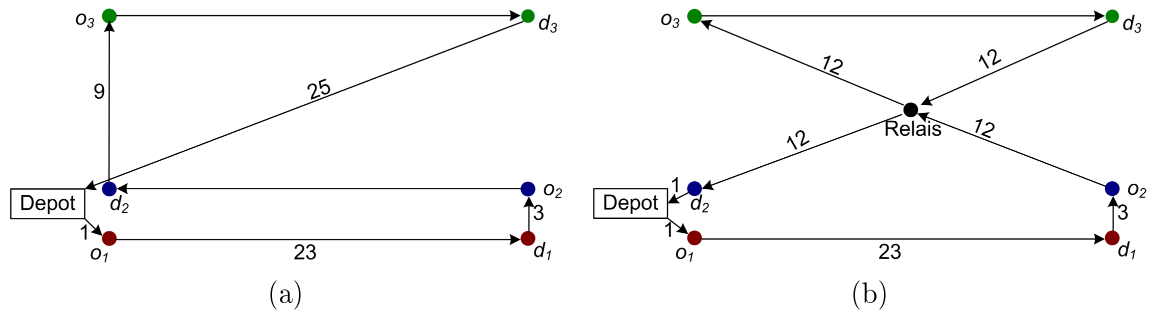


Fig. 3.2 – (a) Solution sans relais, coût = 107 ; (b) solution avec relais pour le même problème, coût = 99

de sa NP-complétude en utilisant une réduction à partir du problème du Voyageur de Commerce (*Traveling Salesman Problem*, TSP). Ils fournissent également un schéma d'approximation en 9/5 pour ce problème.

A notre connaissance, le *stacker crane problem* préemptif a été peu étudié. Atallah et Kosaraju ([AK88]) ont été les premiers à traiter le cas préemptif pour le SCP symétrique. Ils ont étudié à la fois le cas préemptif et non préemptif dans le cas où le graphe sous-jacent est un chemin élémentaire ou un cycle élémentaire. Ils ont prouvé que, dans les deux cas, le problème se résout en un temps polynomial. Frederickson et Guan ([FG92], [FG93]) ont étudié les versions préemptives et non préemptives du SCP symétrique dans le cas où le graphe sous-jacent est un arbre. Ils ont montré que dans ce cas, le problème est NP-complet et ont fourni plusieurs α-approximations. Enfin, une analyse polyédrale du *stacker crane problem* préemptif a été réalisée par Lacroix [Lac09].

Plusieurs variantes de problème de *pickup and delivery* proches du SCP ont été étudiées. On peut mentionner tout d'abord le problème du Voyageur de Commerce avec ramassage et livraison (*Traveling Salesman Problem with Pickup and Delivery*, TSPPD) qui correspond au SCP non préemptif et dans lequel il n'y a pas de contrainte de capacité. Dans [RBL02], les auteurs comparent trois schémas heuristiques de résolution pour le TSPPD. Dans [DRCL10], les auteurs modélisent le TSPPD à l'aide d'un programme linéaire et donnent une analyse de sa structure polyédrale. Ils utilisent des algorithmes de *branch and cut* pour la résolution d'instances allant jusqu'à 35 demandes.

Timlin et Pulleyblank ([TP92]) proposent également un problème proche du SCP dans lequel les sommets sont partitionnés en plusieurs sous-ensembles, l'origine et la destination de chaque demande appartiennent au même sous-ensemble. Le problème consiste à déterminer une solution dans laquelle les sous-ensembles sont visités l'un après l'autre. Les auteurs proposent deux approches heuristiques.

[AJR00], [BFP95] [GP06] considèrent la version asymétrique du TSP dans laquelle sont ajoutées des contraintes de précédence sur les nœuds. Dans certaines publications cette variante a été nommée *Sequential Ordering Problem* (SOP). [AEGS93] s'intéresse à un modèle 0-1 et à la détermination de borne inférieure pour certaines instances inspirées de problèmes réels. Dans [AJR00], les auteurs donnent le programme linéaire associé à ce problème. Ils ont résolu de manière exacte des instances allant jusqu'à 200 nœuds à l'aide d'un algorithme de *branch and cut*. Dans [BFP95], les auteurs proposent une approche

de résolution polyédrale. [GP06] introduit de nouvelles inégalités pour lesquelles des techniques de coupe sont développées et utilisées conjointement à des algorithmes polynomiaux pour les séparer. Des approches heuristiques ont été étudiées pour résoudre ce problème : on peut citer [GD00] et [MSG08] qui utilisent un algorithme de colonie de fourmis.

D'autres problèmes de type TSPPD avec des contraintes supplémentaires ont été étudiés. [CILSG10] considère une variante du TSPPD avec une politique de chargement de type LIFO. [CMC10] étudie le problème avec transbordements (le transport d'une demande est effectué par un premier véhicule de l'origine de la demande jusqu'à un nœud de déchargement, puis la demande est rechargée par un second véhicule et transportée jusqu'à destination). Enfin, [MML06] propose un TSPPD avec transbordement et fenêtre de temps. Hernández-Pérez et González [HPSG09] se sont intéressés à une généralisation du TSP appelée *multi-commodity one-to-one Pickup-and-Delivery TSP* (m-PDTSP) dans laquelle les clients demandent plusieurs types de marchandises et le véhicule est de capacité limitée.

D'autre part, il peut être intéressant de voir les similitudes entre le *stacker crane problem*, qui fait partie des problèmes *pickup and delivery* et le *Hoist Scheduling Problem* (HSP) qui fait partie des problèmes d'ordonnancement. Le *hoist scheduling problem* concerne les lignes de production dotées d'une ressource de transport. L'objectif est d'ordonnancer les mouvements d'un robot de manutention. Dans ce problème des pièces métalliques doivent subir un traitement chimique, pour cela elles sont plongées dans différentes cuves d'acide dans un ordre donné. Chaque traitement a une durée comprise entre une durée minimale et une durée maximale. Le transfert d'une pièce d'une cuve à l'autre est assuré par un robot élévateur de capacité unitaire (une description plus précise peut être trouvée dans [MB03]). Ces robots ont donc les mêmes contraintes (transport d'un unique objet d'un point à un autre) que les grues portiques dont s'inspire le SCP. Le SCP peut alors être vu comme le problème d'ordonnancement des mouvements d'un robot du HSP.

Nous nous intéressons ici à la version préemptive et asymétrique du *stacker crane problem*. Tout d'abord, nous donnons une formulation du problème, puis nous montrons qu'il est possible de modéliser des tournées sous forme d'arbres. Cette représentation permet à la fois une résolution par programmation linéaire en nombres entiers, par des heuristiques de construction et par des recherches locales. Ces dernières fournissent des résultats numériques de très bonne qualité qui sont présentés en section 3.2.5.

3.2.2 Le *Stacker Crane Problem* Preemptif et Asymétrique (SCPPA)

Nous proposons dans cette section une description formelle d'un problème et d'une solution du SCPPA. Le but est de caractériser les tournées qui peuvent être solution du SCPPA. Cette caractérisation nous permet, par la suite, de proposer une modélisation à l'aide d'arbres et de reformuler le problème comme un problème de recherche d'arbre spécifique.

3.2.2.1 Description du SCPPA

Le SCPPA peut se définir de la manière suivante :

- étant donné un graphe G, un véhicule V effectue une tournée dans G afin de satisfaire un ensemble K de demandes. Chaque demande $k \in K$ est exprimée à l'aide d'un couple (o_k, d_k) de nœuds de G, tel que o_k est l'origine de la demande k, d_k est la destination de la demande k, et V doit transporter exactement une unité de charge de o_k vers d_k ;

- l'ensemble des nœuds $X = \{0, 1, \ldots, n\}$ est tel que le nœud 0 représente le dépôt (noté *Depot*) et les autres nœuds sont : soit l'origine d'une demande, soit la destination. Les arcs du graphe G sont orientés et valués par des coûts ;

- le véhicule V possède une capacité égale à 1 ;

- le véhicule V est autorisé à traiter une demande k de manière préemptive : lors du transport de sa charge C il peut s'arrêter à un nœud x quelconque du réseau, décharger C, traiter une autre demande, puis revenir en x recharger C. Un tel nœud x est appelé *relais* pour la demande k. Une demande peut être déchargée plusieurs fois ;

- V doit commencer et finir sa tournée au nœud 0 qui représente le dépôt et noté *Depot* ;

- le coût de la tournée effectuée par V est donné par la somme des coûts des arcs traversés par V.

Dans la suite, nous avons besoin de pouvoir identifier les nœuds par leur fonction (dépôt, origine, destination ou relais), les nœuds origine et destination d'une demande pouvant être utilisés comme relais, on duplique ces nœuds afin de les rendre tous différents (d'un point de vue logique). Ainsi, on peut décomposer X de la manière suivante : $X = \{Depot\} \cup X_O \cup X_D \cup X_R$ où :

$X_O = \{o_k, k \in K\}$;
$X_D = \{d_k, k \in K\}$;
X_R contient une copie de *Depot*, X_O et X_D et, éventuellement, un ensemble de relais.

De manière habituelle, on considère que G est un graphe complet (quitte à le compléter par un calcul des plus courts chemins). Par conséquent, sa description se ramène à une matrice de distances. On note $d(x, x')$ la distance entre deux nœuds x et x'. On suppose que les distances satisfont les inégalités triangulaires et que $d(x, x') = 0$ si x' est une copie de x.

3.2.2.2 Description d'une solution du SCPPA

La caractérisation des solutions que nous proposons s'appuie, entre autre, sur une structure de liste. Nous présentons tout d'abord les principales notations utilisées concernant la manipulation de listes. Ensuite, nous proposons de représenter une tournée comme une liste d'objets nommés *liens labellisés*. Enfin, nous caractérisons les listes de liens labellisés qui représentent une solution du SCPPA.

3.2.2.2.1 Notations préliminaires sur les listes

Soit $L = \{x_1, \ldots, x_n\}$ une liste constituée d'objets $x_i, i \in \{1, \ldots, n\}$.

- Pour tout élément $x = x_i$ de L, nous notons $Succ_L(x)$ (resp. $Pred_L(x)$), le successeur x_{i+1} (resp. prédécesseur x_{i-1}) de x dans L, et par $Rg_L(x)$ le rang de x dans L. Une tournée constituée d'un unique élément x est notée $\{x\}$, et une tournée vide est notée $\emptyset$. Le nombre d'éléments de L est noté $|L|$. Le $i^{ème}$ élément de L est désigné par L_i ;

- On appelle *sous-tournée* de L une tournée L' telle que $L' = \{x_{i_1}, \ldots, x_{i_p}\}$ avec $i_1 < \cdots < i_p$ ($p \in \mathbb{N}$, $i_p \leq n$) ;

- On note $\oplus$ l'opérateur de *concaténation* qui transforme deux séquences $L = \{x_1, \ldots, x_n\}$ et $L' = \{y_1, \ldots, y_n\}$ en une unique séquence $L \oplus L' = \{x_1, \ldots, x_n, y_1, \ldots, y_n\}$;

- On appelle *coupe* de L une décomposition de L en une concaténation $L' \oplus L''$;

- Soient x_i et x_j deux éléments de L tels que $i = Rg_L(x_i) \leq j = Rg_L(x_j)$. On appelle *segment* de L la sous-liste $\{x_i, \ldots, x_j\}$ de L qui est définie par tous les z tels que $i \leq Rg_L(z) \leq j$ et on note $S_L(x, y)$ le segment dans L qui commence en x et finit en y.

3.2.2.2.2 Liens labellisés

Nous proposons de représenter le déplacement du véhicule V d'un nœud x à un nœud y grâce à un *lien labellisé* : un lien labellisé est un triplet $r = (x, y, k)$, où x et y sont dans l'ensemble X et k est dans l'ensemble $\{0\} \cup K$, x (resp. y) est noté $Origine(r)$ (resp. $Dest(r)$) et k est appelé le label de r, noté $Label(r)$. La signification de k est alors que V est vide si $k = 0$ et contient la charge associée à la demande k sinon. Une tournée définie sur X peut alors être représentée comme une suite Γ de liens labellisés. Le coût d'une tournée Γ est donné par :

$$C_{tour}(\Gamma) = \sum_{r \in \Gamma} d(Origine(r), Dest(r))$$

Soient une tournée Γ et un label k, on note Γ^k la suite des liens labellisés qui dérive naturellement de Γ en considérant dans Γ uniquement les liens labellisés r tels que $Label(r) = k$.

La figure 3.3 représente la tournée Γ définie par la suite de liens labellisés : { ($Depot$, o_1, 0), (o_1, x, 1), (x, o_2, 0), (o_2, y, 2), (y, o_3, 0), (o_3, x, 3), (x, d_1, 1), (d_1, y, 0), (y, d_2, 2), (d_2, 0, x), (x, d_3, 3), (d_3, $Depot$, 0)}. Les valeurs sur les arcs indiquent l'ordre de parcours. Le tableau 3.1 donne les distances entre les nœuds de la tournée Γ de manière à pouvoir calculer son coût $C_{tour}(\Gamma) = 271$. A partir de Γ, on peut former les sous-tournées :

Γ^1 = { (o_1, x, 1), (x, d_1, 1) } ;
Γ^2 = {(o_2, y, 2), (y, d_2, 2)} ;
Γ^3 = {(o_3, x, 3),(x, d_3, 3)}.

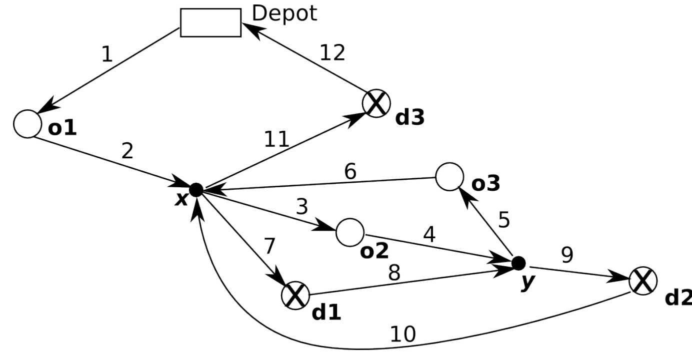


FIG. 3.3 – Exemple de tournée Γ, $C_{tour}(\Gamma) = 271$ (les distances sont données par le tableau 3.1)

	0	o_1	o_2	o_3	d_1	d_2	d_3	x	y
0	0	23	27	30	32	50	20	19	43
o_1	23	0	37	43	35	71	39	20	57
o_2	27	37	0	13	9	33	15	18	19
o_3	30	43	13	0	21	25	11	29	12
d_1	32	35	9	21	0	39	23	17	25
d_2	50	71	33	25	39	0	36	51	14
d_3	20	39	15	11	23	36	0	23	24
x	19	20	18	29	17	51	23	0	37
y	43	57	19	12	25	14	24	37	0

TAB. 3.1 – Matrice des distances associée à la figure 3.3

3.2.2.2.3 Tournées valides

Une suite quelconque de liens labellisés ne représente pas forcément une solution du SCPPA. Il faut pour cela qu'elle représente le déplacement d'un véhicule qui traite toutes les demandes $k \in K$ de manière appropriée. Une tournée Γ définie par une suite de liens labellisés qui représente une solution du SCPPA est appelée tournée *valide*, ce qui signifie que :

- pour chaque couple (r, r') de liens labellisés consécutifs dans Γ, on a $Dest(r) = Origine(r')$;
- l'origine du premier lien labellisé dans Γ (Γ_1) est le nœud *Depot*, la destination du dernier lien labellisé ($\Gamma_{|\Gamma|}$) est également le nœud *Depot* ;
- chaque nœud $x \in X_O \cup X_D$ appartient à 2 liens labellisés r et r' où r' est le successeur de r dans Γ ;
- le nœud *Depot* appartient uniquement à Γ_1 et $\Gamma_{|\Gamma|}$;
- pour chaque demande $k \in K$ la sous-tournée Γ^k associée est telle que :
  1. l'origine du premier lien labellisé dans Γ^k est o_k ;
  2. la destination du dernier lien labellisé dans Γ^k est d_k ;
  3. pour chaque couple (r, r') de liens labellisés consécutifs dans Γ^k on a $Dest(r) = Origine(r')$.

Notons que la tournée de la figure 3.3 est une tournée valide.

Le SCPPA peut alors être posé de la manière suivante : « étant donnés l'ensemble des nœuds X et la matrice des distances, calculer une tournée valide Γ de coût minimal. »

3.2.3 Proposition de modélisation d'une tournée à l'aide d'un arbre

Le but de cette section est de montrer que certaines tournées (y compris les tournées optimales) peuvent être représentées par un arbre. Cette représentation permet d'intégrer aisément l'hypothèse de préemption, les contraintes de capacité et facilite la manipulation des tournées. Elle est donc très pratique pour définir des heuristiques et voisinages (voir section 3.2.4).

Les tournées qui peuvent être représentées par un arbre possèdent certaines propriétés. L'utilisation de la représentation en arbre réduit donc l'espace de recherche, il faut alors s'assurer que cette restriction nous ne empêche pas de trouver les tournées optimales. Le théorème de restriction énonce les propriétés des tournées qui peuvent être représentées en arbre et montre qu'il est possible de restreindre la recherche de solutions optimales à ces tournées.

3.2.3.1 Théorème de restriction

Afin d'introduire le théorème de restriction, nous définissons quelques notations. Considérons Γ une tournée valide. Pour chaque lien labellisé $r = (x, o_k, 0)$ dans Γ, nous notons $\sigma_\Gamma(r)$ l'unique lien labellisé $(d_k, y, 0)$ également dans Γ ; r représente alors l'arrivée du véhicule à l'origine de la demande k et $\sigma_\Gamma(r)$ représente le départ du véhicule de la destination de la demande k, le véhicule étant vide dans les deux cas.

De même, si x est un nœud de X_R tel qu'il existe un lien labellisé $r = (y, x, k)$, $k \geq 1$, $y \in X$ dans Γ, alors on note $\sigma_\Gamma(r)$ le premier lien labellisé $r' = (x, z, k), z \in X$ qui existe dans Γ^k. Remarquons que $\sigma_\Gamma(r)$ n'est défini que pour certains liens labellisés r (ceux dont la destination est : soit l'origine d'une demande, soit un relais).

On dira que deux liens labellisés r et r', pour lesquels $\sigma_\Gamma(r)$ et $\sigma_\Gamma(r')$ sont définis, *se chevauchent* si :

$$Rg_\Gamma(\sigma_\Gamma(r')) > Rg_\Gamma(\sigma_\Gamma(r)) > Rg_\Gamma(r') > Rg_\Gamma(r)$$

Par exemple, dans la figure 3.3 si on prend $r = (o_1, x, 1)$ et $r' = (x, o_2, 0)$ alors $\sigma_\Gamma(r) = (x, d_1, 1)$ et $\sigma_\Gamma(r') = (d_2, x, 0)$. De plus, $Rg_\Gamma(r) = 2$, $Rg_\Gamma(r') = 3$, $Rg_\Gamma(\sigma_\Gamma(r)) = 7$ et $Rg_\Gamma(\sigma_\Gamma(r')) = 10$. Donc ces deux liens labellisés se chevauchent. Ici, il y a chevauchement car V se rend à l'origine de la demande 2 après être passé à l'origine de la demande 1 (la charge de la demande 1 ayant été déposée au nœud relais) mais ne satisfait pas la demande 2 avant la demande 1. La figure 3.4 illustre ce chevauchement en représentant chronologiquement le passage de V aux différents nœuds.

Ces définitions permettent d'énoncer le *théorème de restriction* qui établit que l'on peut restreindre la recherche de solutions optimales pour le SCPPA à des tournées valides dotées de certaines propriétés additionnelles. Ces propriétés permettent de représenter les tournées sous forme d'arbre.

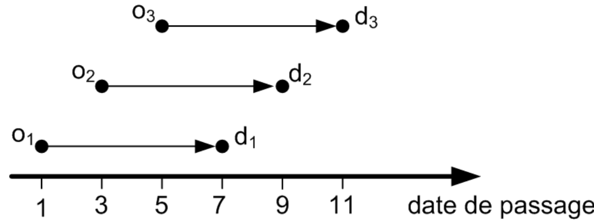


FIG. 3.4 – Ordre chronologique de passage de V aux différents nœuds pour traiter les demandes

Théorème 6 (Théorème de restriction)
Soit Γ *une tournée optimale pour le SCPPA, choisie de telle manière que :*

(A) *:* $|\Gamma|$ *est le plus petit possible ;*

(B) *: Le nombre de liens labellisés* r *dans* Γ *tels que* $Label(r) \neq 0$*, est le plus petit possible, (A) étant supposé satisfait.*

Alors les 4 assertions suivantes doivent être satisfaites :

$(P1)$ *:* Γ *ne contient pas deux occurrences du même lien labellisé* $r = (x, y, k)$*,* $(x, y) \in X$ *et* $k \neq 0$ *;*

$(P2)$ *:* Γ *ne contient pas deux liens labellisés consécutifs* r *et* r' *tels que* $Label(r) = Label(r')$ *;*

$(P3)$ *:* Γ *ne contient pas deux liens labellisés* r *et* r' *qui se chevauchent ;*

$(P4)$ *:* Γ *ne contient pas deux liens labellisés* r *et* r' *tels que* $Dest(r) = Dest(r')$ *et qui soient tous deux de label non nul.*

Démonstration

Rappelons que $r = (x, y, k)$, représente le déplacement du véhicule de x vers y avec la charge associée à la demande k où à vide si $k = 0$.

On suppose donnée une tournée Γ, solution optimale du SCPPA, telle que (A) et (B) sont vrais.

Vérification de $(P1)$

Soit $r = (x, y, k)$, avec $k > 0$. Si r apparaît deux fois dans Γ avec les rangs s et s' alors x et y sont tous les deux des nœuds relais. On peut alors remplacer la valeur du label par 0, dans n'importe quel lien labellisé r'' tel que :

1. $s \leq Rg_\Gamma(r'') < s'$;
2. $Label(r'') = k$.

On obtient alors une tournée valide qui est une solution optimale du SCPPA ce qui amène une contradiction avec l'hypothèse (B).

Vérification de $(P2)$

Si $r = (x, y, k)$ et $r' = (y, z, k)$ sont deux liens labellisés consécutifs dans Γ tels que $Label(r) = Label(r') = k$, on peut alors les remplacer par un unique lien labellisé (x, z, k). Comme les distances respectent les inégalités triangulaires, on obtient une solution optimale ce qui contredit l'hypothèse (A).

Vérification de $(P3)$

Pour démontrer $(P3)$, on va montrer qu'il est possible de changer l'ordre de parcours de la tournée, de manière à pouvoir supprimer un arc, sans que cela n'ait d'incidence sur les demandes. On aura donc une contradiction avec l'hypothèse (A).

Supposons qu'il existe deux liens labellisés $r = (x, y, k)$ et $r' = (x', y', k')$ dans Γ qui se chevauchent. S'il existe dans Γ plusieurs couples (r, r') de liens labellisés qui se chevauchent, on choisit celui qui est tel que $Rg_\Gamma(\sigma_\Gamma(r')) - Rg_\Gamma(r)$ est le plus petit possible (H_1). Les nœuds x et x' doivent alors être des nœuds relais : si x était un nœud origine o_h alors on aurait $k' \neq h$ et il existerait un lien labellisé r'' tel que :

1. $Label(r'') = h$;
2. $Rg_\Gamma(r) < Rg_\Gamma(r'') < Rg_\Gamma(r') < Rg_\Gamma(\sigma_\Gamma(r'')) < Rg_\Gamma(\sigma_\Gamma(r)) < Rg_\Gamma(\sigma_\Gamma(r'))$.

Alors r'' et r' se chevauchent également ce qui induit une contradiction avec (H_1). De la même manière, on peut vérifier que x' n'est pas un nœud origine. x et x' sont donc des nœuds relais et il est clair que $k \neq k' \neq 0$. On peut donc écrire

$r = (y, x, k)$
$r' = (y', x', k')$
$\sigma_\Gamma(r) = (x, z, k)$
$\sigma_\Gamma(r') = (x', z', k')$

On pose :

$S_1 = S_\Gamma(\Gamma_1, r)$
$S_2 = S_\Gamma(Succ_\Gamma(r), r')$
$S_3 = S_\Gamma(Succ_\Gamma(r'), Pred_\Gamma(\sigma_\Gamma(r)))$
$S_4 = S_\Gamma(\sigma_\Gamma(r), Pred_\Gamma(\sigma_\Gamma(r')))$
$S_5 = S_\Gamma(\sigma_\Gamma(r'), \Gamma_{|\Gamma|})$

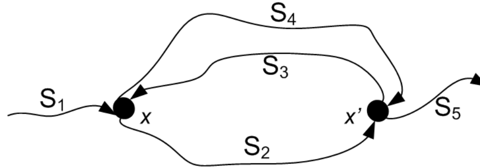


FIG. 3.5 – Les tournées définies par $\Gamma = S_1 \oplus S_2 \oplus S_3 \oplus S_4 \oplus S_5$ et $\Gamma_{bis} = S_1 \oplus S_4 \oplus S_3 \oplus S_2 \oplus S_5$ ont le même nombre d'éléments et le même coût

On remplace Γ par $\Gamma_{bis} = S_1 \oplus S_4 \oplus S_3 \oplus S_2 \oplus S_5$. Il est clair que Γ et Γ_{bis} ont le même nombre d'éléments et le même coût (voir figure 3.5). Il faut de plus, pour conclure cette démonstration, que Γ_{bis} soit une tournée valide, c'est l'objet du lemme suivant :

Lemme 1

Γ_{bis} est une tournée valide.

Démonstration du lemme

Il faut montrer que permuter S_2, S_3 et S_4 n'induit pas de changement dans les Γ^k, c'est-à-dire que pour $k \in K$, on a $\Gamma^k = \Gamma^k_{bis}$. Si le contraire était vrai on pourrait trouver $k'' \neq k'$, k et k'' non nuls et deux liens labellisés r'' et $\sigma_\Gamma(r'')$ de label k" ou avec la destination de r'' égale à $o_{k''}$ de telle manière que une des trois relations suivantes soit vraie :

– $r'' \in S_2$ et $\sigma_\Gamma(r'') \in S_3$ (H_2)
– $r'' \in S_2$ et $\sigma_\Gamma(r'') \in S_4$ (H_3)

– $r'' \in S_3$ et $\sigma_\Gamma(r'') \in S_4$ (H_4)

Si (H_2) ou (H_3) est vraie r'' et r' se chevauchent ce qui contredit l'hypothèse (H_1). Si (H_4) est vrai r et r'' se chevauchent ce qui contredit l'hypothèse (H_1). On peut conclure que Γ_{bis} est une tournée valide.

Ce lemme permet de conclure la démonstration de (P3) en remarquant que r et $\sigma_\Gamma(r)$ sont consécutifs dans la tournée valide Γ_{bis} ce qui implique (démonstration de $(P2)$) que r et $\sigma_\Gamma(r)$ peuvent être remplacés dans Γ_{bis} par un unique lien labellisé (y, z, k) de telle sorte que le coût de Γ_{bis} n'augmente pas et que son nombre d'éléments décroît ce qui contredit l'hypothèse (A).

Vérification de $(P4)$

Intuitivement, on utilise la même démarche que pour démontrer $(P3)$.

Supposons que Γ contient deux liens labellisés r et r' qui sont tels que $Dest(r) = Dest(r')$ et tels que $Rg_\Gamma(r) < Rg_\Gamma(r')$. Puisque le nœud x origine de r ne peut pas être dans $\{Depot\} \cup X_D$ (sinon le label de r serait nul) il appartient à X_R et est utilisé deux fois comme nœud relais. Donc r, $\sigma_\Gamma(r)$, r', $\sigma_\Gamma(r')$ peuvent être notés :

$r = (y, x, k), k \neq 0$
$r' = (y', x, k'), k' \neq 0, k$
$\sigma_\Gamma(r) = (x, z, k)$
$\sigma_\Gamma(r') = (x, z', k')$

(P3) implique :

$Rg_\Gamma(r) < Rg_\Gamma(\sigma_\Gamma(r)) < Rg_\Gamma(r') < Rg_\Gamma(\sigma_\Gamma(r'))$ $\quad (H_5)$

ou

$Rg_\Gamma(r) < Rg_\Gamma(r') < Rg_\Gamma(\sigma_\Gamma(r')) < Rg_\Gamma(\sigma_\Gamma(r))$ $\quad (H_6)$

Supposons (H_5) et posons :

$S_1 = S_\Gamma(\Gamma_1, r)$
$S_2 = S_\Gamma(Succ_\Gamma(r), Pred_\Gamma(\sigma_\Gamma(r)))$
$S_3 = S_\Gamma(\sigma_\Gamma(r), r')$
$S_4 = S_\Gamma(Succ_\Gamma(r'), Pred_\Gamma(\sigma_\Gamma(r')))$
$S_5 = S_\Gamma(\sigma_\Gamma(r'), \Gamma_{|\Gamma|})$

et remplaçons Γ par $\Gamma_{bis} = S_1 \oplus S_3 \oplus S_2 \oplus S_4 \oplus S_5$. Il est clair que Γ et Γ_{bis} ont le même nombre d'éléments et le même coût. De plus, on peut montrer comme précédemment que Γ_{bis} est une tournée valide. Γ_{bis} peut être raccourci en remplaçant r et $\sigma_\Gamma(r)$ par un unique lien labellisé (y, z, k) tel que le coût de Γ_{bis} n'augmente pas et son nombre d'éléments diminue, ce qui contredit l'hypothèse (A). On procède de même avec (H_6). □

3.2.3.2 *Arbre-Reformulation* du SCPPA

Le théorème précédent montre qu'il est possible de réduire l'espace de recherche à des tournées valides qui satisfont les propriétés $(P1)$ à $(P4)$, ces tournées sont nommées tournées *fortement valides*. Ainsi résoudre le SCPPA revient à trouver une tournée fortement valide et de coût minimal. La figure 3.6 (a) montre la tournée de la figure 3.3 transformée en tournée fortement valide et la figure 3.6 (b) illustre l'ordre chronologique de passage du véhicule aux différents nœuds pour traiter les demandes. On constate qu'il n'y a plus de

chevauchement.

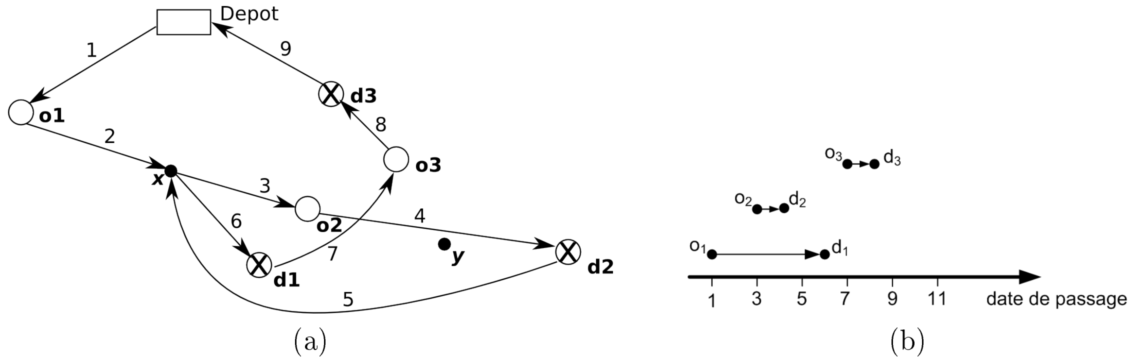


FIG. 3.6 – (a) Tournée de la figure 3.3 transformée en tournée fortement valide ; (b) Ordre chronologique de passage du véhicule aux différents nœuds

Nous montrons maintenant comment une tournée fortement valide peut être représentée comme un arbre. Cette modélisation nous permet de développer des heuristiques de construction et des opérateurs de transformation locale simples et efficaces.

On appelle *arbre biparti ordonné* un arbre $\mathcal{A}$ tel que :

- Ses nœuds peuvent être ordonnés en deux classes A et B de telle manière que la classe A a ses fils dans la classe B et vice versa ;
- Pour chaque nœud x dans $\mathcal{A}$ qui n'est pas une feuille (nœud terminal) l'ensemble des fils associés à x est ordonné linéairement et défini sous forme de liste.

On dit qu'un arbre $\mathcal{A}$ biparti ordonné est *consistant* avec le SCPPA défini par l'ensemble K des demandes et par l'ensemble X des nœuds ((X, K)-consistant) si :

($P5$) Un nœud dans $\mathcal{A}$ peut être identifié soit par une demande $k \in K$ (alors nommé nœud *demande*), soit par un nœud dans $\{Depot\} \cup X_R$ (alors nommé nœud *relais*). Notons que tous les nœuds *demande* existants doivent apparaître dans $\mathcal{A}$ alors qu'il peut n'y avoir que quelques (voire aucun) nœuds parmi les nœuds *relais* dans $\mathcal{A}$. Les nœuds *relais* effectivement dans $\mathcal{A}$ forment l'ensemble des nœuds *relais actifs* de $\mathcal{A}$, noté $Actif(\mathcal{A})$;

($P6$) La racine de $\mathcal{A}$ est le nœud *Depot* et les feuilles sont toutes des nœuds *demande* ;

($P7$) Pour chaque nœud *demande* k, son ensemble de fils linéairement ordonné $Rel(\mathcal{A}, k)$ (qui peut être vide) est composé uniquement de nœuds *relais* et son père est un nœud relais actif ;

($P8$) Pour chaque nœud *relais* x, son ensemble de fils linéairement ordonné $Dem(\mathcal{A}, x)$ est composé de nœuds *demande* et son père est un nœud *demande.*

Pour un tel arbre biparti ordonné $\mathcal{A}$, on peut définir un coût $C_{arbre}(\mathcal{A})$ de la façon suivante :

- Pour chaque nœud *demande* $k \in K$, on pose :

$Rel_1(\mathcal{A},k)$ le premier élément dans la liste des relais fils de k et $Rel_{fin}(\mathcal{A},k)$ le dernier élément dans la liste des relais fils de k, alors

Si k est une feuille $C(k) = dist(o_k, d_k)$,

sinon $C(k) = dist(o_k, Rel_1(\mathcal{A},k)) + dist(Rel_{fin}(\mathcal{A},k), d_k) + \sum_{\substack{x \in Rel(\mathcal{A},k),\\ x \neq Rel_{fin}(\mathcal{A},k)}} dist(x, Succ_{Rel(\mathcal{A},k)}(x))$;

- Pour chaque nœud $x \in \{Depot\} \cup X_R$, on pose :
  $Dem_1(\mathcal{A},x)$ le premier élément dans la liste des demandes filles de x et $Dem_{fin}(\mathcal{A},x)$ le dernier élément dans la liste des demandes filles de x, alors
  $C(x) = dist(x, o_{Dem_1(\mathcal{A},x)}) + dist(d_{Dem_{fin}(\mathcal{A},x)}, x) + \sum_{\substack{k \in Dem(\mathcal{A},x),\\ k \neq Dem_{fin}(\mathcal{A},x)}} dist(d_k, o_{Succ_{Dem(\mathcal{A},x)}(k)})$;
- Finalement le coût de $\mathcal{A}$ s'écrit : $C_{arbre}(\mathcal{A}) = \sum_{k \in K} C(k) + \sum_{x \in Actif(\mathcal{A})} C(x)$.

Par exemple, on donne les coûts associés à l'arbre représenté figure 3.7 (b).
Le coût des demandes s'écrit :

$C(1) = dist(o_1, x) + dist(x, d_1)$;
$C(2) = dist(o_2, d_2)$;
$C(3) = dist(o_3, d_3)$.

Le coût des relais s'écrit :

$C(x) = dist(x, o_2) + dist(d_2, x)$;
$C(Depot) = dist(Depot, o_1) + dist(d_3, Depot) + dist(d_1, o_3)$.

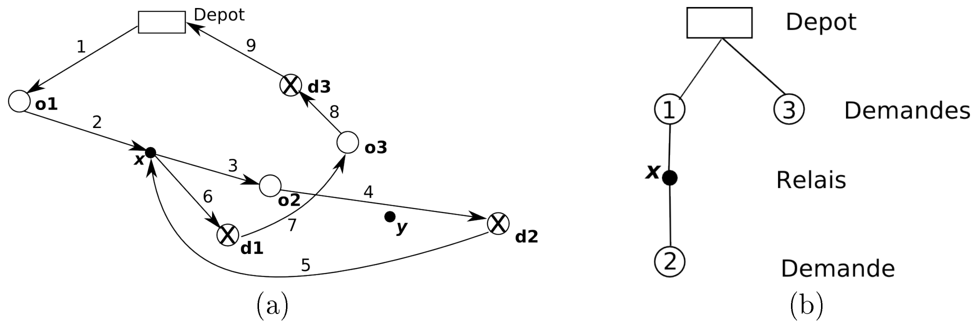


FIG. 3.7 – (a) Tournée fortement valide ; (b) Arbre associé

Théorème 7
Il y a une bijection nommée Arbre entre les tournées fortement valides et les arbres bipartis ordonnés (X, K)-consistants. De plus, pour toute tournée fortement valide Γ associée à un arbre $\mathcal{A}$ on a $C_{arbre}(\mathcal{A}) = C_{tour}(\Gamma)$.

Démonstration

Soit Γ une tournée fortement valide, on obtient l'arbre associé $\mathcal{A} = Arbre(\Gamma)$ de la manière suivante :

- $Actif(\mathcal{A})$ est défini par l'ensemble des nœuds de $\{Depot\} \cup X_R$ qui appartiennent aux liens labellisés de Γ ;
- Pour chaque nœud demande $k \in K$ l'ensemble de ses fils $Rel(\mathcal{A},k)$ est fait des

nœuds relais qui apparaissent dans les liens labellisés de Γ^k, ordonnés dans l'ordre où ils apparaissent dans Γ^k ;

- Pour chaque nœud relais x de $\{Depot\} \cup X_R$, on pose $\rho(x) = (x, y, 0)$ et $\tau(x) = (z, x, 0)$ les deux liens labellisés de label nul qui contiennent x dans Γ et qui sont tels que $Rg_\Gamma(\rho(x)) < Rg_\Gamma(\tau(x))$. On en déduit l'ensemble des demandes filles $Dem(\mathcal{A}, x)$ du relais x : une demande $k \in K$ est fille de x si l'unique lien labellisé $r^k = (o_k, t, k)$ qui apparaît dans Γ est tel que
  1. $Rg_\Gamma(\rho(x)) < Rg_\Gamma(r^k) < Rg_\Gamma(\tau(x))$;
  2. il n'existe pas de nœuds relais y tel que $Rg_\Gamma(\rho(x)) < Rg_\Gamma(\rho(y)) < Rg_\Gamma(r^k) < Rg_\Gamma(\tau(y)) < Rg_\Gamma(\tau(x))$.

A partir d'une telle construction, vient l'équivalence du théorème 7. □

Une tournée fortement valide et l'arbre biparti associé sont représentés figure 3.7. Notons que les fils d'une demande correspondent aux sommets *relais* sur lesquels celle-ci est déchargée. Les fils d'un sommet *relais* actif x correspondent aux demandes transportées (jusqu'à leur destination finale) sur le circuit partant de x.

Corollaire 1
Résoudre le SCPPA revient à trouver un arbre biparti ordonné $\mathcal{A}$ consistant avec X, K et tel que $C_{arbre}(\mathcal{A})$ est le plus petit possible.

L'intérêt de ce dernier résultat est de nous fournir une formulation du SCPPA grâce à des arbres bipartis. Cette formulation transforme le problème initial de tournées en un problème de construction d'arbre. Elle nous permet également de fournir un programme linéaire pour le SCPPA.

3.2.3.3 Formulation du SCPPA avec un programme linéaire en nombres entiers

Les définitions précédentes permettent de formuler un programme linéaire pour le SCPPA. Pour la formulation de ce problème, on a besoin de séparer les nœuds relais pour distinguer le cas où un nœud relais sert à décharger et le cas où il sert à recharger. De même, on souhaite distinguer le cas où on sort du dépôt pour débuter la tournée et le cas où on y rentre à la fin de la tournée. Nous construisons donc tout d'abord un graphe auxiliaire $G = (X^*, E)$:

$X^* = X \cup X_R^* \cup \{Depot^*\}$, où X_R^* est une copie de X_R et $Depot^*$ est une copie de $Depot$;
Pour chaque nœud x dans X_R, on note x^* sa copie dans X_R^*. De même, pour chaque origine $x = o_k$ dans X_O, on note x^* le nœud d_k associé dans X_D.

$E = \{(Depot, x), x \in X_O\} \cup \{(x, Depot^*), x \in X_D\} \cup \{(o_k, d_k), k \in K\} \cup \{(d_{k'}, o_k), k \neq k' \in K\} \cup \{(x, y), (y, x), x \in X_O, y \in X_R\} \cup \{(x, y)(y, x), x \in X_R^*, y \in X_D\} \cup \{(x, y), x \in X_R^*, y \in X_R\}$.

Chaque arc $e \in E$ possède une longueur $dist^*(e)$.

Nous rappelons qu'un chemin ω d'un tel graphe G est une suite de nœuds telle que, pour chaque nœud x dans ω, le couple $(x, Succ_\omega(x))$ définit un arc de E. Une tournée fortement valide Γ peut être transformée en un chemin γ de G de sorte que :

$(P9)$: γ commence en *Depot* et finit en *Depot** et γ est un chemin élémentaire c'est-à-dire qu'il visite chaque nœud au plus une fois ;

$(P10)$: pour chaque demande $k \in K$, γ visite o_k et d_k dans cet ordre, et pour chaque relais $x \in X_R$, γ visite x si et seulement s'il visite x^*, et dans ce cas, il le fait dans cet ordre ;

$(P11)$: pour chaque couple (x, y), $x \neq y$, dans $X_R \cup X_O$ l'implication suivante doit être vraie : $\Big(Rg_\gamma(x) < Rg_\gamma(y) \text{ et } Rg_\gamma(y) < Rg_\gamma(x^*)\Big) \Rightarrow Rg_\gamma(y^*) < Rg_\gamma(x^*)$
On appelle cette condition la *condition de non chevauchement* ;

$(P12)$: $C_{tour}(\Gamma) = \sum\limits_{\substack{x\in\gamma,\\ x\neq Depot^*}} dist^*(x, Succ_\gamma(x)).$

Un chemin ω de G qui vérifie les propriétés $(P9)$ à $(P12)$ est dit *fortement valide.*

Théorème 8
Pour tout chemin fortement valide ω de G, il existe une tournée Γ fortement valide telle que $\gamma = \omega$, où γ est le chemin de G associé à Γ.

Démonstration

Pour démontrer ce théorème, nous donnons tout d'abord l'algorithme de construction de Γ à partir de ω (voir algorithme 33).

Dans cet algorithme l'instruction (I_1) est valide car on a $(P10)$ et parce qu'un arc de G qui arrive en x_2 doit venir d'un nœud origine ou d'un nœud de X_R^*. Dans ce cas, il est clair que $k \neq 0$.

Nous obtenons le résultat du théorème en procédant par récurrence sur la longueur (nombre de nœuds) de ω. Si ω ne contient pas de nœud dans X_R le résultat est immédiat. Sinon, considérons $x_0 \in X_R$ le premier nœud de X_R qui apparaît dans ω. $Pred_\omega(x_0)$ doit donc être un nœud origine $o_k, k \in K$. Par conséquent, l'arc $(Pred_\omega(x_0), Succ_\omega(x_0^*))$ appartient à E et le fait de supprimer le segment $S_\omega(x_0, x_0^*)$ de ω fournit un autre chemin ω_1 de G. On pose :

$K_1 = \{k \in K\}$, tel que o_k et d_k sont dans ω_1 ;

$K_2 = \{k \in K\}$, tel que o_k et d_k sont dans $S_\omega(x_0, x_0^*)$.

K_1 et K_2 définissent une partition de K, et $S_\omega(x_0, x_0^*)$ peut être vu comme un chemin fortement valide si on considère que l'ensemble K est restreint à K_2 et que x_0 joue le rôle du dépôt. Donc, avec cette hypothèse, on peut écrire $\omega_2 = \gamma_2$. De la même manière ω_1 est aussi un chemin fortement valide si on considère que l'ensemble K est restreint à K_1 et peut donc être écrit $\omega_1 = \gamma_1$. Il ne reste qu'à insérer γ_2 entre $(Pred_\omega(x_0), x_0, k)$ et $(x_0, Succ_\omega(x_0^*), k)$ dans γ_1 pour obtenir Γ tel que $\omega = \gamma$. $\square$

Algorithme 33 : construction de la tournée Γ à partir du chemin ω

Entrées : ω
Sorties : Γ

```
1  x ← Depot;
2  Γ ← ∅;
3  Tant que x ≠ Depot* faire
4  |  y ← Succ_ω(x);
5  |  Si il existe un nœud z ∈ {Depot} ∪ X_R tel que y = z* alors
6  |  |  y_2 = z
7  |  sinon
8  |  |  y_2 = y
9  |  suivant x faire
10 |  |  cas où x = Depot
11 |  |  |  Γ ← {(x, y, 0)} ⊕ Γ
12 |  |  cas où x = o_k, k ∈ K
13 |  |  |  Γ ← {(x, y, k)} ⊕ Γ
14 |  |  cas où x ∈ X_R
15 |  |  |  Γ ← {(x, y_2, 0)} ⊕ Γ
16 |  |  cas où x ∈ X_D
17 |  |  |  Γ ← {(x, y_2, 0)} ⊕ Γ
18 |  |  cas où x ∈ X_R*
19 |  |  |  Γ ← {(x, y, k)} ⊕ Γ, où k ≠ 0 est tel que le lien labellisé
   |  |  |  (Pred_ω(x_2), x_2, k) (où x_2 est tel que x_2* = x) est déjà dans ω (I_1)
20 |  x ← y;
```

Corollaire 2
Résoudre une instance (X, K) *du SCPPA revient à trouver un chemin* γ *fortement valide de longueur minimale dans* G.

Théorème 9
Résoudre un SCPPA (X, K) *revient à résoudre le programme linéaire* $\mathcal{P}$ *donné figure 3.8.*

Démonstration

Il est possible de représenter un chemin γ fortement valide de manière à établir un programme linéaire en nombres entiers. Le modèle linéaire donné par la figure 3.8 implique un vecteur flot $z = (z_e, e \in E)$ à valeur dans $\{0, 1\}$, un vecteur rang $R = (R_x, x \in X)$ à valeurs entières et un vecteur de décision t à valeur dans $\{0, 1\}$ indexé sur les couples (x, y), $x \neq y, x, y \in X_O \cup X_R$. Ces vecteurs ont la sémantique suivante :

$\forall e \in E, z_e = 1$ ssi l'arc e est dans γ ;
$\forall x \in \gamma, R_x$ est égal au rang de x dans γ ;
$\forall (x, y), x \neq y, (x, y) \in X_R \cup X_O$:

$$t_{x,y} = 1 \text{ ssi } Rg_\gamma(y) < Rg_\gamma(x^*) ;$$

$$t_{x,y} = 0 \text{ ssi } Rg_\gamma(x^*) < Rg_\gamma(y).$$

Pour obtenir le résultat du théorème 9, il suffit d'appliquer le corollaire 2 en écrivant les propriétés $(P9)\dots(P12)$ avec le formalisme des modèles linéaires. On obtient alors le programme linéaire de la figure 3.8. □

$$
\mathcal{P}:\left\{\begin{array}{lll}
\textbf{Minimiser} \sum_{e\in E} dist^*(e).z_e & & \\
\sum_{x\in X^*} z_{(x,Depot)} = 0 & & (C1)\\
\sum_{x\in X^*} z_{(x,Depot^*)} = 1 & & (C2)\\
\sum_{x\in X^*} z_{(Depot,x)} = 1 & & (C3)\\
\sum_{x\in X^*} z_{(Depot^*,x)} = 0 & & (C4)\\
\forall x \in X_D \cup X_O, & \sum_{y\in X^*} z_{yx} = 1 & (C5)\\
\forall x \in X_R \cup X_{R^*}, & \sum_{y\in X^*} z_{yx} \leq 1 & (C6)\\
\forall x \in X_R, & \sum_{y\in X^*} z_{(yx)} = \sum_{y\in X^*} z_{(yx^*)} & (C7)\\
\forall k \in K, & R_{o_k} \leq R_{d_k} - 1 & (C8)\\
\forall x \in X_R, & R_x \leq R_{x^*} - 1 & (C9)\\
\forall x \in X^* - \{Depot, Depot^*\}, & \sum_{y\in X^*} z_{(yx)} - \sum_{y\in X^*} z_{(xy)} = 0 & (C10)\\
\forall e = (x,y) \in E, & z_e + (R_x + 1 - R_y)/|X^*| \leq 1 & (C11)\\
\forall (x,y), x \neq y, x,y \in X_R \cup X_O, & t_{x,y} + (R_y + 1 - R_{x^*})/|X^*| \leq 1 & (C12)\\
\forall (x,y), x \neq y, x,y \in X_R \cup X_O, & t_{x,y} + (R_y - 1 - R_{x^*})/|X^*| \geq 0 & (C13)\\
\forall (x,y), x \neq y, x,y \in X_R \cup X_O, & t_{x,y} + (R_{y^*} + 1 - R_{x^*})/|X^*| \leq 1 & (C14)
\end{array}\right.
$$

FIG. 3.8 – Programme linéaire pour le SCPPA

Les contraintes du programme linéaire $\mathcal{P}$ ont la signification suivante :

(C1) aucun arc n'a pour destination *Depot* ;
(C2) un unique arc a pour destination *Depot** ;
(C3) un unique arc a pour origine *Depot* ;
(C4) aucun arc n'a pour origine *Depot** ;
(C5) chaque sommet x de $X_D \cup X_O$ est destination d'un unique arc ;
(C6) chaque sommet x de $X_R \cup X_{R^*}$ est destination d'au plus un arc ;
(C7) un sommet x de X_R est visité si et seulement si le sommet correspondant x^* dans X_{R^*} est visité ;
(C8) le rang de l'origine d'une demande est inférieur au rang de la destination ;
(C9) le rang du relais de type déchargement est inférieur au rang du relais rechargement correspondant ;

(C10) traduit les lois de Kirchhoff ;
(C11) traduit l'implication $z_e = 1 \Rightarrow R_x + 1 - R_y \leq 0$: s'il existe un arc (x, y) alors le rang de x est inférieur au rang de y ;
(C12) traduit l'implication $t_{x,y} = 1 \Rightarrow R_y + 1 - R_{x^*} \leq 0$ (I_1) ;
(C13) traduit l'implication $t_{x,y} = 0 \Rightarrow R_y - 1 - R_{x^*} \geq 0$ (I_2) ;
(C14) traduit l'implication $t_{x,y} = 1 \Rightarrow R_{y^*} + 1 - R_{x^*} \leq 0$ (I_3).

Dans les contraintes *(C12)* à *(C14)*, le vecteur t sert à traduire la condition de non chevauchement. Les deux premières inégalités (I_1) et (I_2) donnent la sémantique de t : $t_{x,y} = 1$ signifie $R_y < R_{x^*}$ (I_1) et $t_{x,y} = 0$ signifie $R_{x^*} < R_y$ (I_2). La troisième inégalité (I_3) interdit la configuration suivante : $R_x < R_y < R_{x^*} < R_{y^*}$ dans laquelle on a chevauchement. En effet, si $R_y < R_{x^*}$, il vient de (I_3) que $R_{y^*} < R_{x^*}$. De même, en inversant les rôles de x et de y, on obtient $R_x < R_{y^*} \Rightarrow R_{x^*} < R_{y^*}$. Donc, si on a $R_y < R_{x^*}$ (I_3) donne que les seuls rangs possibles pour y^* et x sont tels que : $R_y < R_{y^*} < R_x < R_{x^*}$ ou $R_x < R_y < R_{y^*} < R_{x^*}$.

Remarquons également que (I_1) et (I_3) sont redondantes. En effet, si $R_{y^*} < R_{x^*}$ alors $R_y < R_{x^*}$ car par définition $R_y < R_{y^*}$.

Notons que les contraintes *(C1)* à *(C6)* traduisent la propriété $(P9)$, les contraintes *(C7)* à *(C9)* traduisent la propriété $(P10)$ et les contraintes *(C12)* à *(C14)* traduisent la propriété $(P11)$.

3.2.4 Heuristiques à base d'arbres pour le SCPPA

Les algorithmes décrits dans cette section se déduisent facilement de la modélisation à l'aide d'arbres des solutions du SCPPA obtenues précédemment. Ces algorithmes sont des algorithmes de construction et des algorithmes de descente basés sur deux types d'opérateurs : opérateurs d'insertion et opérateurs de transformation locale. Afin d'alléger le texte, on appelle simplement *relais* un nœud *relais* et *demande* un nœud *demande*.

3.2.4.1 Construction d'une solution initiale

Les opérateurs d'insertion sont définis sur un arbre biparti ordonné $\mathcal{A}$, consistant avec l'ensemble des nœuds X et avec un sous-ensemble K' de l'ensemble des demandes K. $\mathcal{A}$ est alors un arbre partiel comprenant uniquement les demandes de K'. Ces opérateurs insèrent une demande $k \in K \setminus K'$ dans $\mathcal{A}$. On utilise deux opérateurs :

Insertion-simple(k, x, i_d) : cet opérateur insère la demande k dans la liste des demandes filles du relais actif x en position i_d ;

Insertion-avec-relais(k, y, k', i_x) : cet opérateur insère tout d'abord le relais y non actif dans la liste des relais fils de la demande $k' \in K'$ en position i_x, y devient donc actif. Ensuite, la demande k est insérée comme fille de y.

La figure 3.9 illustre l'utilisation de *Insertion-simple* et la figure 3.10 illustre l'utilisation de *Insertion-avec-relais*.

La figure 3.11 montre les tournées schématisées (ne respectent pas les distances) associées à chacun des arbres des figures 3.9 et 3.10.

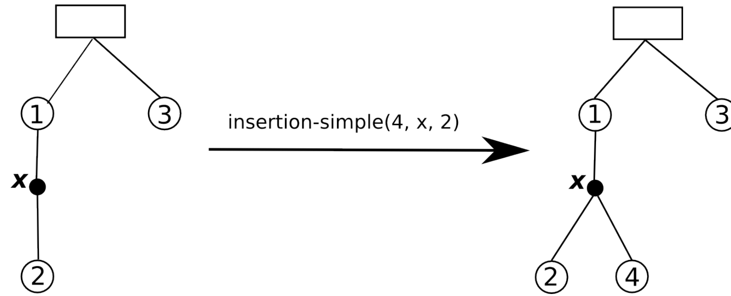


FIG. 3.9 – Insertion de la demande 4 dans la liste des demandes filles du relais x en position 2

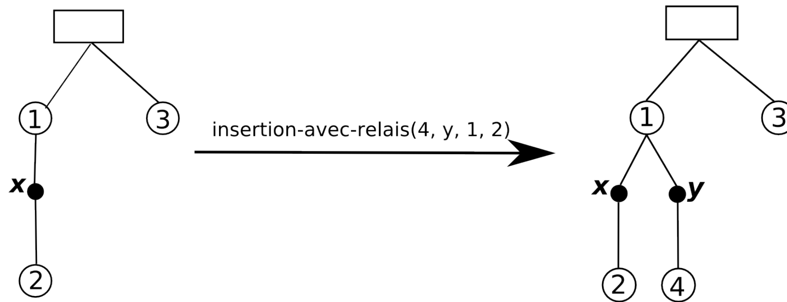


FIG. 3.10 – Insertion de la demande 4 après le relais y lui même inséré dans la liste des relais de la demande 1 en position 2

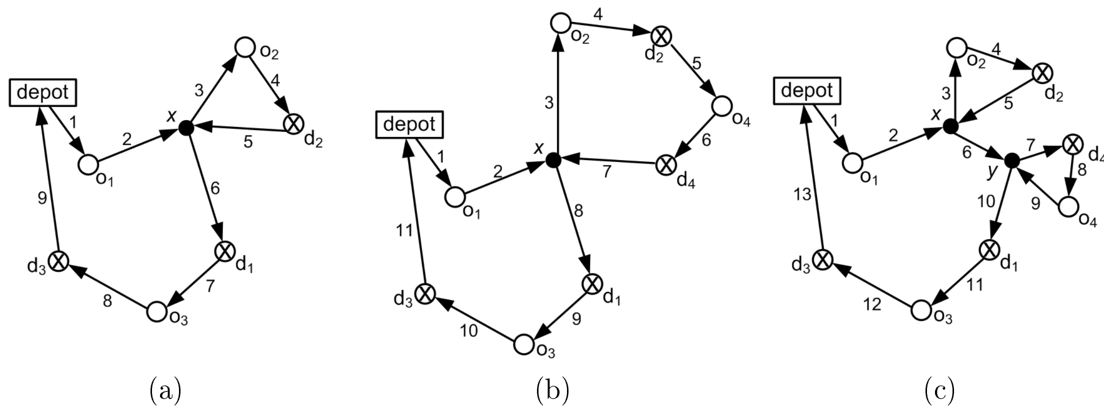


FIG. 3.11 – Tournées schématisées associées aux arbres des figures : (a) 3.9 avant insertion, (b) 3.9 après insertion et (c) 3.10 après insertion

Nous pouvons alors proposer un algorithme glouton `SCPPA-Insertion` (voir l'algorithme 34), basé sur la représentation en arbre des tournées, en utilisant les opérateurs d'insertion. L'arbre $\mathcal{A}$ est tout d'abord initialisé à la racine $\{Depot\}$. Les demandes sont insérées séquentiellement (dans un ordre aléatoire), en choisissant à chaque fois la meilleure insertion possible dans l'arbre partiel $\mathcal{A}$ au sens de $C_{arbre}(\mathcal{A})$. Il est clair qu'il est inutile de tester toutes les positions d'insertion possibles avec tous les relais possibles pour une nouvelle demande. En effet on peut calculer a priori les relais qui semblent les plus prometteurs pour chaque demande : on calcule l'ensemble des relais x « voisins » (qui minimisent $d(x, o_k) + d(d_k, x)$) de chaque demande k. On ne considère uniquement, au cours de l'algo-

rithme, que les trois meilleurs voisins. L'algorithme 34 étant non déterministe on l'exécute plusieurs fois et on garde la meilleure solution obtenue (voir algorithme 35).

Algorithme 34 : SCPPA-Insertion

Sorties : $\mathcal{A}$

1. Définir aléatoirement un ordre ρ sur les demandes de K ;
2. $\mathcal{A} = \{Depot\}$;
3. **Pour** $k \in K$ suivant l'ordre ρ **faire**
4.   **Choisir** un opérateur d'insertion I et le jeu de paramètres u associé tels que l'insertion de k dans $\mathcal{A}$ via $I(u)$ induise une augmentation de $C_{arbre}(\mathcal{A})$ la plus faible possible ;
5.   **Appliquer** $I(u)$ à $\mathcal{A}$;

Algorithme 35 : Multi-SCPPA-Insertion

Entrées : $N \in \mathbb{N}$

Sorties : $\mathcal{A}_{best}$

1. $\mathcal{A}_{best} = \emptyset$;
2. $C_{best} = +\infty$;
3. **Pour** $i = 1$ *à* N **faire**
4. $\mathcal{A}$ = `SCPPA-Insertion` ;
5.   **Si** $C_{arbre}(\mathcal{A}) < C_{best}$ **alors**
6. $\mathcal{A}_{best} = \mathcal{A}$;
7. $C_{best} = C_{arbre}(\mathcal{A})$;

3.2.4.2 Recherche locale de type VNS

La recherche locale utilise des opérateurs de transformation qui agissent sur un arbre biparti ordonné $\mathcal{A}$, consistant avec l'ensemble des nœuds X et l'ensemble des demandes K et ils le modifient. On définit six opérateurs :

Remplace-relais(x, y) : cet opérateur remplace le relais actif x par le relais non actif y dans $\mathcal{A}$ (y devient donc actif et x non actif);

Deplace-relais$(k_1, k_2, i_1, i_2, i_3)$: cet opérateur supprime de la liste des fils de la demande k_1 le segment allant de i_1 à i_2 et l'insère dans l'ensemble des fils de la demande k_2 à partir de la position i_3. On suppose, pour utiliser cet opérateur, que k_1 ne *domine* pas k_2 dans $\mathcal{A}$, c'est-à-dire que k_2 ne doit pas appartenir à la branche partant de k_1 ;

Rearrange-relais(k, i_1, i_2, i_3) : dans la liste des fils de la demande k, cet opérateur déplace le segment allant de i_1 à i_2 pour le mettre à la position i_3 ;

Deplace-demande$(x_1, x_2, j_1, j_2, j_3)$: cet opérateur supprime de la liste des fils du relais x_1 le segment allant de j_1 à j_2 et l'insère dans l'ensemble des fils du relais x_2 à partir de la position j_3. Si $Dem(\mathcal{A}, x)$ devient vide, le relais x est alors supprimé de $\mathcal{A}$ et devient non actif. On suppose, pour utiliser cet opérateur, que x_1 ne domine pas x_2 dans $\mathcal{A}$;

Rearrange-demande$(x,\ j_1,\ j_2,\ j_3)$: dans la liste des fils du relais x, cet opérateur déplace le segment allant de j_1 à j_2 pour le mettre à la position j_3 ;

Deplace-demande-relais$(x,\ y,\ k,\ j_1,\ j_2,\ i)$: dans la liste des fils du relais x, on supprime le segment S allant de j_1 à j_2, on insère y à la position i dans la liste des fils de la demande k, y devient donc actif, enfin S est inséré dans $Dem(\mathcal{A}, y)$. Si $Dem(\mathcal{A}, x)$ devient vide alors x devient non actif. On suppose, pour utiliser cet opérateur, que la demande k n'est dominée par aucune des demandes dans S.

Les figures 3.12 à 3.17 illustrent l'application des opérateurs de transformation locale sur un arbre contenant 9 demandes et 5 relais. L'opérateur *Deplace-demande-relais* est le seul à permettre l'ajout d'un nouveau relais dans l'arbre, le résultat des tests présentés dans la section 3.2.5 a permis de mettre en évidence l'efficacité de cet opérateur.

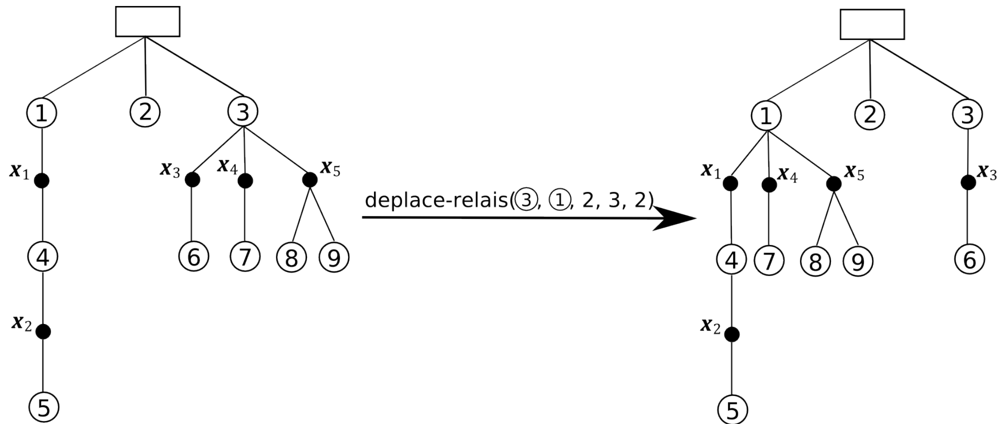


FIG. 3.12 – Utilisation de l'opérateur *Remplace-relais*, le relais actif x_1 est remplacé par le relais non actif y

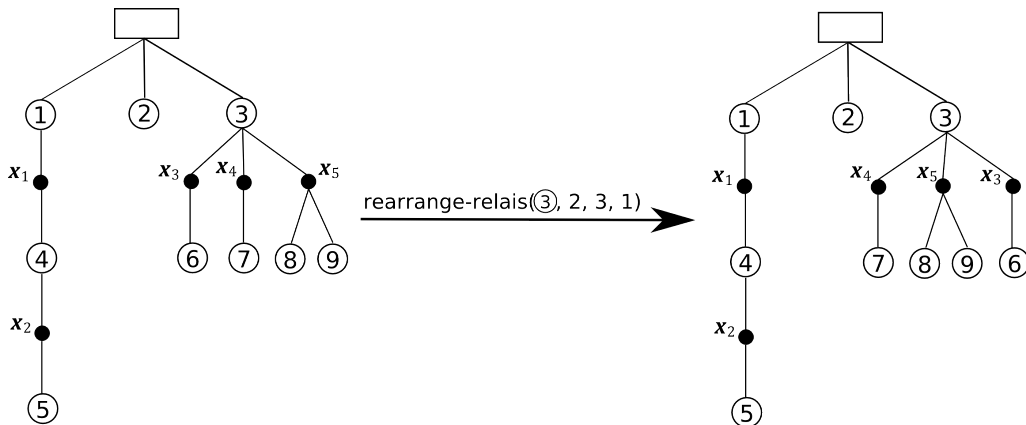


FIG. 3.13 – Utilisation de l'opérateur *Deplace-relais*, les relais actifs x_4 et x_5 sont déplacés

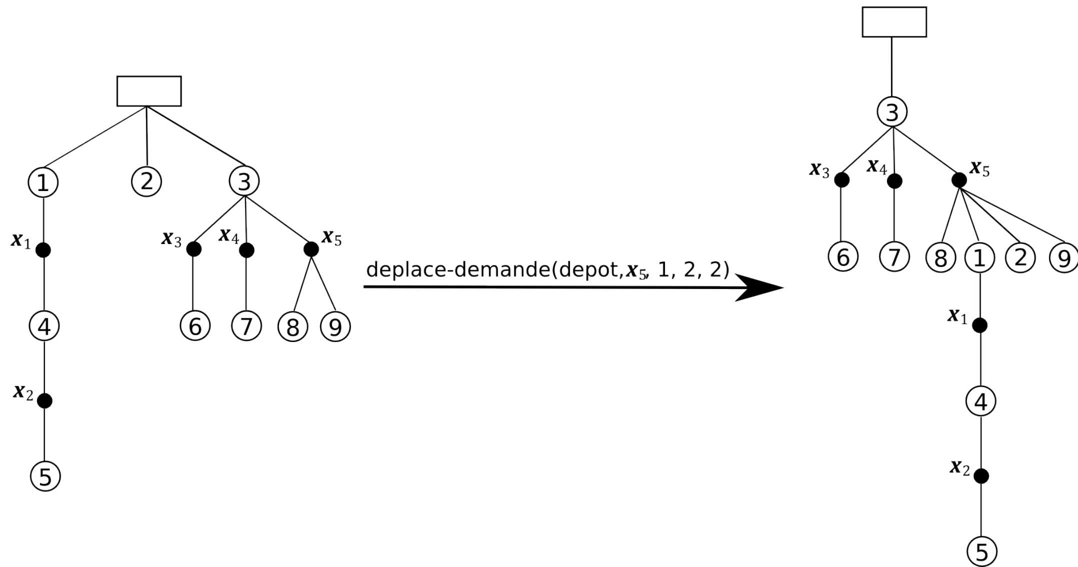


FIG. 3.14 – Utilisation de l'opérateur *Rearrange-relais* sur les relais fils de la demande 3

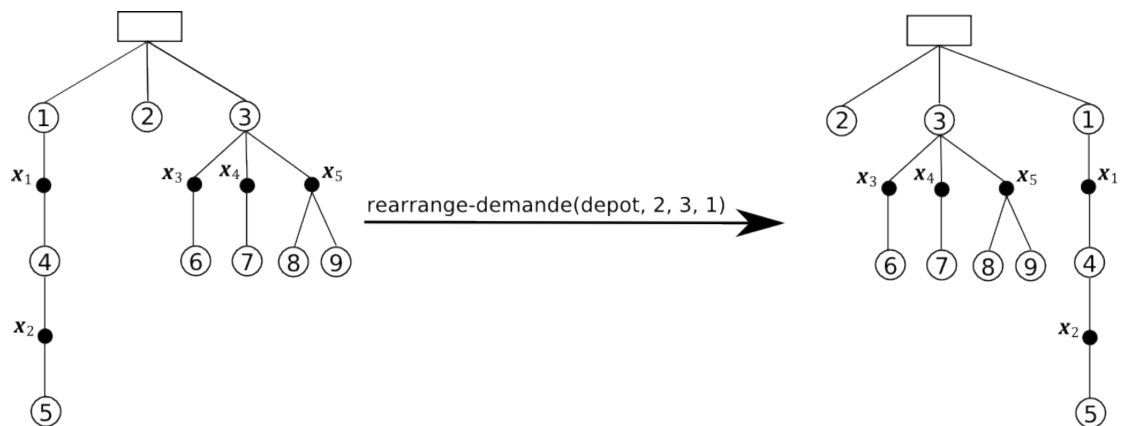


FIG. 3.15 – Utilisation de l'opérateur *Deplace-demande*, les demandes 1 et 2 sont déplacées

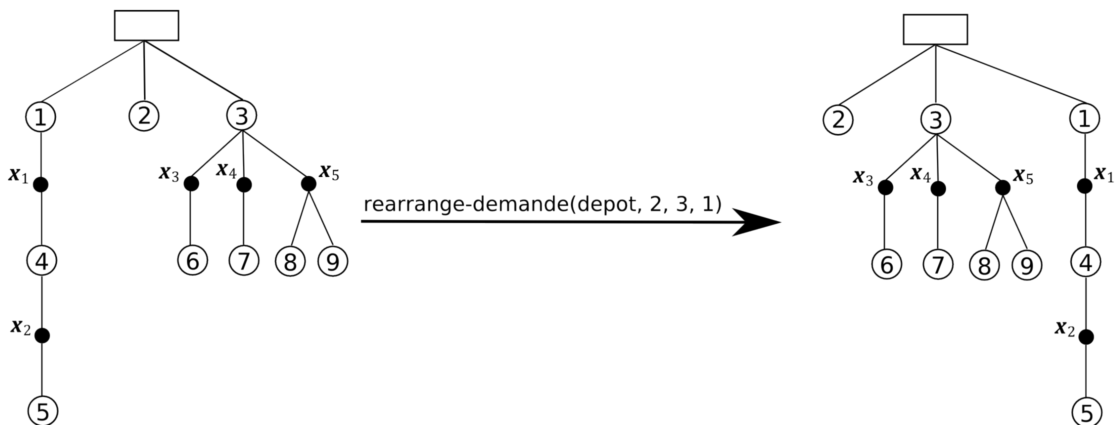


FIG. 3.16 – Utilisation de l'opérateur *Rearrange-demande* sur les demandes filles du dépôt

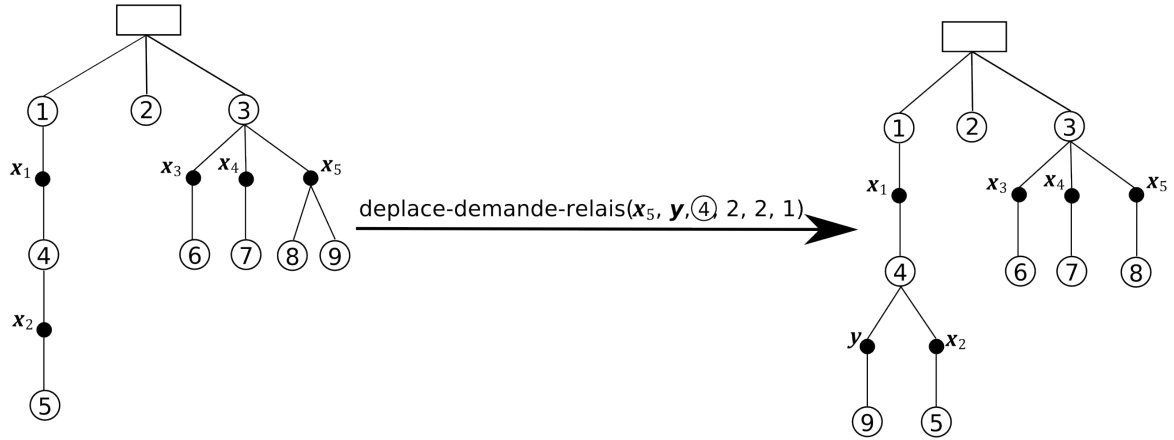


FIG. 3.17 – Utilisation de l'opérateur *Deplace-demande-relais*, le relais y est ajouté dans l'arbre et la demande 8 est déplacée pour devenir fille de y

Les opérateurs de transformation locale nous permettent de construire un algorithme de descente `SCPPA-Descente` décrit par l'algorithme 36 : il agit sur un arbre $\mathcal{A}$ préalablement construit à l'aide de `SCPPA-Insertion` et cherche à l'améliorer (c'est-à-dire réduire $C_{arbre}(\mathcal{A})$) en appliquant des transformations locales successives à l'aide des opérateurs décrits précédemment. L'efficacité de l'algorithme dépend en grande partie du choix de l'opérateur à appliquer et de son jeu de paramètres. Le nombre de choix envisageables étant très grand, on évite de tester tous les paramètres possibles pour chaque opérateur : on utilise donc un mécanisme de filtrage similaire à celui proposé dans le cadre de `SCPPA-Insertion`. On introduit de plus une valeur seuil H qui permet de limiter le domaine de recherche des paramètres : le domaine de recherche des paramètres est d'autant plus grand que H est petit. La valeur de H décroît alors au fur et à mesure des itérations permettant ainsi d'agrandir l'espace de recherche à chaque fois que la recherche dans l'espace restreint par H a échoué.

Algorithme 36 : SCPPA-Descente

Entrées : $\mathcal{A}$, H_{init}, H_{min}
Sorties : $\mathcal{A}$

1 $H \leftarrow H_{init}$;
2 stop $\leftarrow$ faux ;
3 **Tant que** *stop* $=$ *faux* **faire**
4   **Rechercher** (en utilisant le filtre H) un opérateur I et un jeu de paramètres u tels que l'application de $I(u)$ sur $\mathcal{A}$ réduise $C_{arbre}(\mathcal{A})$;
5   **Si** *la recherche a échoué* **alors**
6         $H \leftarrow H/2$;
7   **Si** $H < H_{min}$ **alors**
8   stop $\leftarrow$ vrai ;

3.2.5 Résultats expérimentaux

Les algorithmes décrits dans la section 3.2.4 ont été implémentés en C++ (le compilateur utilisé est celui de Visual Studio 2008), testés sur PC Intel Xeon, 1.86GHz, 3.25 GO de Ram.

Deux aspects ont été analysés :
- la capacité de `SCPPA-Insertion` et `SCPPA-Descente` à trouver de manière rapide des solutions proches de l'optimal ;
- l'impact de l'hypothèse de préemption : nombre de relais dans la solution.

Dans ce but, plusieurs sortes de tests ont été réalisés. Les instances testées ont été générées à partir des instances de la TSPLIB (disponibles sur `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`). Ces dernières fournissent les ensembles de nœuds X et les distances entre chaque nœud. Les couples origine / destination ont été choisis aléatoirement parmi les nœuds de X. Les instances traitées comportent de 20 à 300 nœuds et de 10 à 100 couples origine / destination. Les instances de taille modérée ont été résolues de manière exacte (sur un PC AMD Opteron, 2.1GHz) par un algorithme de coupes et branchements (voir [Lac09]) à l'aide d'un programme linéaire similaire à celui donné dans la section 3.2.3.

3.2.5.1 Performance de l'algorithme glouton

Les résultats obtenus avec $N = 100$ exécutions de l'algorithme de construction `SCPPA-Insertion` sont présentés dans les tableaux 3.2 pour des instances à 46 sommets, 3.3 pour des instances à 94 sommets et 3.4 pour des instances à 238 sommets. Dans chaque tableau, dix instances sont présentées.

Nous utilisons les notations suivantes :
- *dem.* : nombre de demandes ;
- *som.* : nombre de sommets ;
- *inst.* : nom de l'instance ;
- *ref* : valeur de la solution optimale obtenue grâce au programme linéaire ;
- *min* : valeur de la meilleure solution obtenue parmi les $N = 100$ itérations de `SCPPA-Insertion` ;
- *écart* : écart en % entre *ref* et *min* ;
- *rel.* : nombre moyen de relais actifs dans les solutions produites par `SCPPA-Insertion` ;
- *dem/rel* : nombre moyen de demandes dans les listes filles de relais (sauf dépôt)
- *cpu* : temps total en millisecondes des N exécutions de `SCPPA-Insertion`.

Le tableau 3.2, concernant les instances à 11 demandes et 46 sommets, montre la rapidité d'exécution de notre algorithme de construction sur des instances de petite taille : moins de 15 millisecondes pour 100 exécutions. De plus, on constate que les solutions trouvées sont très proches des solutions optimales : en moyenne, sur les 10 instances, l'écart relatif entre le coût de la solution trouvée par notre algorithme et le coût de la solution optimale est de 0.73%.

dem.	*som.*	*ref.*	*min.*	*écart (%)*	*rel.*	*dem/rel*	*cpu (ms)*
11	46	24 654	25 023	1.50	0.06	2.83	< 15
11	46	21 395	21 424	0.14	0.43	1.60	< 15
11	46	22 834	23 363	2.32	0.36	2.51	< 15
11	46	23 255	23 444	0.81	0.41	2.47	< 15
11	46	23 993	23 993	0.00	0.31	2.03	< 15
11	46	23 233	23 233	0.00	0.54	1.71	< 15
11	46	20 224	20 283	0.29	0.11	1.72	< 15
11	46	20 865	21 124	1.24	0.61	1.13	< 15
11	46	23 054	23 073	0.08	0.43	3.00	< 15
11	46	26 704	26 963	0.97	0.16	4.40	< 15

TAB. 3.2 – Tests de `SCPPA-Insertion` (100 exécutions) sur 10 instances à 46 sommets

dem.	*som.*	*ref.*	*min.*	*écart (%)*	*rel.*	*dem/rel*	*cpu (ms)*
23	94	358 048	375 447	4.86	0.97	1.67	< 15
23	94	280 579	291 800	4.00	1.44	1.39	15
23	94	318 959	325 027	1.90	0.95	3.25	< 15
23	94	315 118	322 908	2.47	0.53	2.46	15
23	94	320 578	327 709	2.22	0.27	2.80	15
23	94	306 000	318 838	4.20	0.59	4.42	< 15
23	94	314 127	330 528	5.22	0.52	3.04	16
23	94	342 390	350 960	2.50	0.53	4.57	16
23	94	337 327	343 127	1.72	0.54	3.24	15
23	94	330 119	339 509	2.84	0.73	1.61	16

TAB. 3.3 – Tests de `SCPPA-Insertion` (100 exécutions) sur 10 instances à 94 sommets

Le tableau 3.3, concernant les instances à 23 demandes et 94 sommets, montre la rapidité d'exécution de notre algorithme de construction sur des instances de taille moyenne : 100 exécutions durent environ 15 millisecondes. De plus, on constate que les solutions trouvées sont assez proches des solutions optimales : en moyenne, sur les 10 instances, l'écart entre le coût de la solution trouvée par notre algorithme et le coût de la solution optimale est de 3.19%.

dem.	*som.*	*ref.*	*min.*	*écart (%)*	*rel.*	*dem/rel*	*cpu (ms)*
59	238	?	327 070	-	0.42	5.13	78
59	238	?	332 679	-	0.64	4.60	78
59	238	?	326 189	-	0.27	8.95	78
59	238	?	366 750	-	0.36	5.74	78
59	238	?	310 179	-	1.53	3.05	78
59	238	?	319 619	-	0.64	3.24	78
59	238	?	285 989	-	0.70	4.46	78
59	238	?	345 100	-	0.56	7.33	78
59	238	?	333 909	-	0.47	11.11	78
59	238	318 099	337 259	6.02	0.36	9.19	78

TAB. 3.4 – Tests de `SCPPA-Insertion` (100 exécutions) sur 10 instances à 238 sommets

Le tableau 3.4, concernant les instances à 59 demandes et 238 sommets, montre les résultats de notre algorithme de construction sur des instances de taille assez grande. On constate tout d'abord que l'algorithme est très rapide : en moyenne 100 exécutions de l'algorithme durent 78 millisecondes. L'algorithme de *branch and cut*, qui nous permet

de trouver les solutions exactes, n'a résolu qu'une instance sur les 10 (la limite de temps autorisée de deux heures ayant été dépassée pour les autres). L'écart entre le coût de la solution trouvée par notre algorithme et le coût de la solution optimale pour cette instance est d'environ 6%.

Le tableau 3.5 montre les résultats de l'algorithme de construction sur des instances *RELn* que nous avons construites de façon à ce que la solution optimale comporte des relais actifs, testant ainsi la capacité de l'algorithme à les trouver. Elles comportent les caractéristiques suivantes :

- l'instance comporte n nœuds, $n/2$ relais et $((n/2)-1)/2$ demandes ;
- sa valeur optimale est $\sum_{k \in K} Dist(o_k, d_k)$.

inst.	*ref.*	*min.*	*écart(%)*	*rel.*	*cpu (ms)*
REL62	11 843	13 045	10.1	0.11	<15
REL110	17 464	19 554	12.0	0.35	16
REL158	21 053	24 235	15.1	0.54	31
REL182	22 323	26 321	17.9	0.54	47
REL230	24 142	28 117	16.5	0.91	78
REL326	26 036	30 734	18.0	1.76	141
REL374	26 494	32 077	21.1	2.87	187
REL422	26 775	32 830	22.6	3.39	250
REL518	27 042	33 773	24.9	5.92	375
REL566	27 098	33 664	24.2	6.86	453

TAB. 3.5 – Tests de `SCPPA-Insertion` effectués sur 10 instances « RELn »

On constate que l'heuristique de construction `SCPPA-Insertion` nous permet d'obtenir des solutions d'assez bonne qualité de manière très rapide. Cependant, on remarque que l'heuristique est de moins en moins efficace à mesure que le nombre de relais actifs augmente.

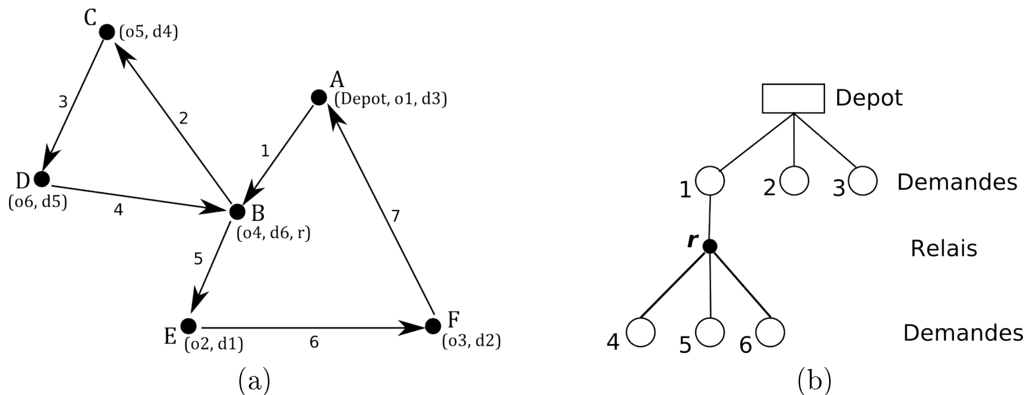


FIG. 3.18 – (a) Tournée solution d'une instance "*RELn*" ; (b) Arbre associé

La figure 3.18 visualise une tournée solution d'une instance de type *RELn* (avec $n = 26$) ainsi que son arbre associé. En chaque point $\{A \dots G\}$ du plan sont positionnés différents nœuds de l'ensemble X : les nœuds origines sont notés $o_i, i = 1 \dots 6$, les nœuds destinations sont notés $d_i, i = 1 \dots 6$ et le nœud relais représenté est l'unique relais actif noté r. Les valeurs sur les arcs de la tournée indiquent l'ordre de parcours.

3.2.5.2 Performance de la recherche locale

Le second test consiste à exécuter une unique fois `SCPPA-Descente` à partir d'une solution initiale fournie par `SCPPA-Insertion` (`SCPPA-Insertion` n'est exécuté qu'une fois pour avoir une solution de qualité moindre et voir l'impact de l'algorithme de descente). On note :

- *dem.* : nombre de demandes ;
- *som.* : nombre de sommets ;
- *ref* : valeur de la solution optimale obtenue grâce au programme linéaire ;
- *min* : valeur de la solution obtenue après l'application `SCPPA-Descente` ;
- *gap init.* : écart en % entre *ref* et la valeur de la solution initiale ;
- *gap fin.* : écart en % entre *ref* et *min* ;
- *amel.* : amélioration en % apportée par la descente (= *gap init.* - *gap fin*) ;
- *transfo.* : nombre de transformations locales effectuées par `SCPPA-Descente` ;
- *rel.* : nombre de relais actifs dans la solution produite par `SCPPA-Descente` ;
- *cpu* : temps total en secondes de l'exécution de `SCPPA-Descente`.

dem.	*som.*	*ref.*	*min.*	*gap init.*	*gap fin.*	*amel.*	*transfo.*	*rel.*	*cpu (s)*
11	46	24 654	24 654	10.3	0.0	10.3	12	1	0.09
11	46	21 395	21 914	5.0	2.4	2.6	3	1	0.05
11	46	22 834	22 874	9.9	0.1	9.8	9	1	0.05
11	46	23 255	23 435	3.9	0.7	3.2	4	2	0.03
11	46	23 993	23 993	0.0	0.0	0.0	0	0	0.03
11	46	23 233	23 763	8.3	2.2	6.1	3	0	0.03
11	46	20 224	20 244	1.2	0.1	0.2	3	1	0.08
11	46	20 865	21 084	4.9	1.0	3.9	6	1	0.03
11	46	23 054	23 054	4.9	0.0	4.9	7	1	0.05
11	46	26 704	26 754	4.9	0.2	3.7	7	1	0.07

Tab. 3.6 – Tests de `SCPPA-Descente` sur 10 instances à 46 sommets

Le tableau 3.6 montre les résultats de la descente sur les instances à 46 sommets. L'écart moyen, sur les 10 instances, à la solution optimale après descente est de 0.6%. L'amélioration moyenne apportée par la descente est de 4.5%.

dem.	*som.*	*ref.*	*min.*	*gap init.*	*gap fin.*	*amel.*	*transfo.*	*rel.*	*cpu (s)*
23	94	358 048	358 349	5.2	0.1	5.1	18	2	0.8
23	94	280 579	284 411	7.9	1.3	6.3	20	4	1.1
23	94	318 959	322 879	5.7	1.2	4.3	12	2	2.0
23	94	315 118	316 362	5.1	0.4	4.6	24	5	0.7
23	94	320 578	323 508	7.1	0.9	6.1	23	1	3.0
23	94	306 000	310 963	12.6	1.6	10.4	38	5	1.4
23	94	314 127	319 009	12.9	1.5	11.1	28	2	1.2
23	94	342 390	342 551	6.7	0.1	6.6	26	4	1.8
23	94	337 327	338 500	6.4	0.3	6.0	35	3	0.9
23	94	330 119	334 881	6.4	1.4	4.8	19	4	0.7

Tab. 3.7 – Tests de `SCPPA-Descente` sur 10 instances à 94 sommets

Le tableau 3.7 montre les résultats de la descente sur les instances à 94 sommets. L'écart moyen, sur les 10 instances, à la solution optimale après descente est de 0.9%. L'amélioration moyenne apportée par la descente est de 6.53%. On constate que sur ces instances de taille moyenne la descente améliore les solutions de manière significative.

dem.	*som.*	*ref.*	*min.*	*gap init.*	*gap fin.*	*amel.*	*transfo.*	*rel.*	*cpu (s)*
59	238	?	313 315	-	-	5.2	89	4	130
59	238	?	318 870	-	-	8.9	116	4	189
59	238	?	314 493	-	-	6.2	96	4	125
59	238	?	360 902	-	-	3.5	107	4	145
59	238	?	293 613	-	-	9.8	99	4	125
59	238	?	303 441	-	-	7.7	90	4	80
59	238	?	265 713	-	-	6.4	84	4	86
59	238	?	330 823	-	-	6.9	77	4	94
59	238	?	314 073	-	-	8.5	105	4	70
59	238	318 099	318 913	9.5	0.25	9.1	96	4	103

Tab. 3.8 – Tests de `SCPPA-Descente` sur 10 instances à 238 sommets

Le tableau 3.8 montre les résultats de la descente sur les instances à 238 sommets. L'amélioration moyenne apportée par la descente est de 7.22%.

inst.	*ref.*	*min.*	*gap init*	*gap fin*	*amel.*	*transfo.*	*rel.*	*cpu (s)*
REL62	11 843	11 953	16.7	0.9	15.8	10	4	0.04
REL110	17 464	17 583	27.0	0.7	36.3	28	8	0.34
REL158	21 053	21 053	29.9	0.0	29.9	56	12	1.5
REL182	22 323	22 605	20.8	1.2	19.6	46	14	2.9
REL230	24 142	24 225	28.4	0.3	28.4	78	18	8.4
REL326	26 036	26 279	19.3	0.9	18.3	79	26	25
REL374	26 494	26 626	26.6	0.5	26.2	112	30	39
REL422	26 775	26 851	33.1	0.3	30.1	134	34	78
REL518	27 042	27 067	37.0	0.1	36.9	162	42	232
REL566	27 098	27 211	33.1	0.4	32.4	202	46	328

Tab. 3.9 – Tests de `SCPPA-Descente` effectués sur 10 instances « RELn »

Le tableau 3.9 montre les résultats de la descente sur les instances « RELn » construites de telle sorte que la solution optimale contienne plusieurs relais. L'écart moyen, sur les 10 instances, à la solution optimale après descente est de 0.53%. L'amélioration moyenne apportée par la descente est de 27.39%. Cette amélioration est très importante, ce qui s'explique par le fait que d'une part, les solutions initiales fournies par l'algorithme de construction sont de qualité moyenne, d'autre part la descente est particulièrement efficace pour créer des relais.

On peut tout d'abord remarquer que l'écart entre préemption et non préemption est très différent suivant la manière dont ont été générées les instances (certaines solutions contiennent beaucoup de relais alors que d'autres n'en ont que très peu voire pas du tout). Bien que notre algorithme de descente `SCPPA-Descente` n'utilise aucun des mécanismes habituels de contrôle destinés à gérer les optima locaux (recherche tabou, recuit simulé

...), on peut constater que les opérateurs qui dérivent de notre représentation en arbre fournissent de très bons résultats dans des temps de calcul très courts. D'autre part, on remarque que la descente locale améliore souvent les solutions initiales en créant des relais. Le rôle de l'opérateur *Deplace-demande-relais* est donc particulièrement important car sur les six opérateurs utilisés, il est le seul à pouvoir créer des relais.

3.2.6 Conclusion

Dans cette section, nous avons traité un problème de *Pickup and Delivery* avec des demandes préemptives et des contraintes de capacité et nous avons montré qu'il est possible de le transformer en un problème de construction d'arbre. La modélisation des solutions sous forme d'arbre permet de construire des heuristiques simples, efficaces et rapides. Elle permet également de créer facilement des opérateurs de transformation locale qui ont été intégrés à une recherche locale de type VNS. Les expérimentations numériques montrent que l'heuristique de construction fournit de bons résultats même si elle a des difficultés à créer des relais. Elles montrent également l'efficacité de la recherche locale. Contrairement à l'heuristique de construction, elle crée facilement des relais et améliore de ce fait nettement les solutions initiales.

Dans la section suivante, nous nous intéressons à un autre type de problème de *pickup and delivery* : le *Dial-a-Ride*. Nous considérons pour ce problème des contraintes additionnelles mettant en jeu des flux financiers entre les demandes.

3.3 *Dial-a-Ride* avec contraintes financières

Cette section s'intéresse aux problèmes de type *Dial-a-Ride* avec contraintes financières. Ce type de contraintes présente un grand intérêt lors de la conception ou de la définition de nouveaux services car il s'agit de proposer des systèmes de tarification adaptés à la demande à un prix acceptable pour les usagers. Notre objectif est de proposer une heuristique pour le *Dial-a-Ride* avec contraintes financières. Ce problème étant une extension au problème de *Dial-a-Ride* « classique », nous divisons cette section en deux parties :

- la première partie concerne l'étude du *Dial-a-Ride* classique. Nous proposons une méthode heuristique pour résoudre ce problème. Une étude numérique montre les performances de cette méthode par rapport aux meilleures méthodes de la littérature ;
- la deuxième partie propose une extension du modèle pour prendre en compte les contraintes financières. Les heuristiques proposées pour le problème classique sont adaptées à ces nouvelles contraintes.

3.3.1 Le *Dial-a-Ride Problem* - Présentation et état de l'art

3.3.1.1 Présentation du problème

Le problème de *Dial-a-Ride* est un problème de transport à la demande. Il fait partie des problèmes de *pickup and delivery*. Il intervient, par exemple, dans le contexte de transports scolaires, transports pour personnes à mobilité réduite...

Dans ce problème, des demandes de transport sont faites auprès d'un opérateur. Chaque demande est associée à une origine (lieu où aller chercher le client) et à une destination (lieu où déposer le client). Pour effectuer ces transports, l'opérateur dispose d'une flotte de véhicules. On distingue le DARP des autres problèmes de *pickup and delivery* car il inclut une notion de qualité de service. Cette qualité de service se traduit généralement par la prise en compte de :

- fenêtres de temps (ce qui permet aux usagers de fixer une date approximative de départ et/ou d'arrivée) ;
- durées maximales de transport ;
- temps d'attente maximaux pour les usagers.

Les problèmes de type *Dial-a-Ride* comprennent de nombreuses contraintes et il n'est pas garanti, a priori, qu'une solution existe. On peut alors chercher une solution minimisant le nombre de violations de contraintes ou la valeur de ces violations. Certains schémas algorithmiques fonctionnent parfois mieux lorsque les contraintes sont relaxées. Dans les deux cas, cela aboutit à inclure dans la fonction objectif une pénalité représentant la violation des contraintes.

3.3.1.2 Etat de l'art

Dans la littérature, le DARP fait référence à un problème de transport à la demande mais il n'existe pas à notre connaissance de définition clairement établie. Ainsi on trouve pour ce problème des définitions qui varient, selon les auteurs, par rapport aux contraintes considérées ou à la fonction objectif.

Le DARP peut être considéré dans le cas statique (toutes les demandes sont connues à l'avance) ou dynamique (les demandes sont connues en temps réel), mono-véhicule (un unique véhicule pour traiter toutes les demandes) ou multi-véhicule. [CL07] fournit une étude sur les différentes formes du DARP ainsi que les principaux modèles et algorithmes.

Nous nous intéressons au cas statique multi-véhicule. Parmi les premiers auteurs à traiter ce cas, on peut citer Jaw *et al.* ([JOPW86]) qui considèrent des fenêtres de temps et une durée maximale de transport pour les usagers, l'objectif étant de minimiser une combinaison non linéaire de critères liés à la qualité de service. Ils résolvent ce problème grâce à une heuristique d'insertion. Le DARP a été fortement étudié ces dernières années, on trouve différentes contraintes et différents objectifs suivant le champ d'application considéré par les auteurs. Par exemple, [XCC06] minimise les coûts d'utilisation des véhicules, les coûts liés aux conducteurs, les temps d'attente et les temps de service. Il considère, par ailleurs, diverses contraintes opérationnelles (fenêtre de temps, véhicules différents qui ne peuvent, suivant leur type, transporter que certains usagers, durées maximales de transport, contraintes relatives aux conducteurs). Au contraire, [BKKS98] ne considère pas les fenêtres de temps comme une contrainte mais comme un objectif : il minimise la violation des fenêtres de temps dans la fonction objectif. Il minimise, de plus, le nombre de véhicules utilisés et des coûts relatifs à la distance parcourue. [TV97] considère une flotte de véhicules hétérogènes, des temps de service, plusieurs dépôts, des fenêtres de temps et une durée de transport maximale. Il minimise uniquement la distance parcourue par les véhi-

cules. Ces différents exemples montrent que la qualité de service peut s'exprimer à travers différentes contraintes. [PCL09] tente d'apporter une définition de la qualité de service et dresse un état de l'art des différents critères qui ont déjà été utilisés dans le cadre du DARP.

De même que [CL03], [Cor06], [PDH10] et [JLB07], nous considérons le DARP avec les contraintes suivantes :
- fenêtre de temps sur l'origine d'une demande et/ou la destination ;
- durée maximale du trajet pour les usagers ;
- véhicules de capacité finie ;
- durée maximale d'une tournée.

Cependant, ces auteurs utilisent des fonctions objectif différentes : [CL03], [Cor06] et [PDH10] minimisent la distance totale parcourue par les véhicules tandis que [JLB07] minimise une combinaison linéaire de la distance et de différentes durées relatives à la qualité de service (durée du trajet, temps d'attente). Il ajoute à cette fonction un coût relatif aux contraintes violées. [PDH10] propose également une version de son algorithme pour prendre en compte la qualité de service dans la fonction objectif. [LS07] et [DD04] considèrent des contraintes similaires, ils minimisent respectivement le nombre de véhicules utilisés et une combinaison linéaire des distances et durées relatives aux trajets.

Ce problème est résolu de façon heuristique par [CL03], [PDH10] et [JLB07] en utilisant respectivement une recherche tabou, une recherche locale à voisinage variable et un algorithme génétique. [Cor06] propose une formulation à l'aide d'un programme linéaire en nombres entiers. Ce programme est résolu de manière exacte grâce à un algorithme de *branch and cut*. Un jeu d'instances pour ce problème a été créé et testé par [CL03]. [PDH10] et [JLB07] l'ont également utilisé. Nous utiliserons donc ces mêmes instances pour nos expérimentations numériques.

3.3.2 Définition

3.3.2.1 Description du problème de *Dial-a-Ride*

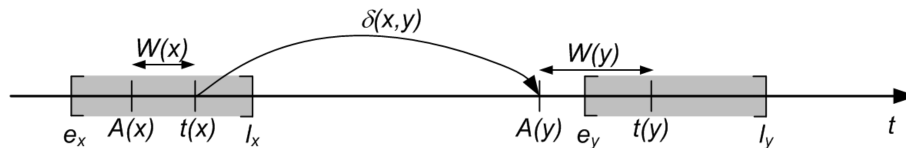
On considère le problème de *Dial-a-Ride* défini par :
- un graphe $G = (X, E)$. $X = \{0, 1 \dots 2n\}$ est l'ensemble des sommets. Le sommet 0 représente le dépôt. Chaque arc (x, y) de E est associé à une distance $d(x, y)$ (distance entre x et y) et à une durée $\delta(x, y)$ (temps que met le véhicule pour aller de x vers y) ;
- un ensemble de K véhicules identiques de capacité Q ;
- un ensemble $D = \{1 \dots n\}$ de n demandes de transport. Chaque demande i est associée à une origine o_i, une destination d_i et une charge q_i (nombre de passagers à transporter) ;
- la charge q_x d'un sommet x de X est positive si x est origine d'une demande, négative si x est destination d'une demande et nulle si x est le dépôt. De plus, pour une même demande i on a $q_{d_i} = -q_{o_i}$.

Le problème contient les contraintes suivantes :

- un sommet x de X est l'origine ou la destination d'une demande. Il est associé à une fenêtre de temps $[e_x, l_x]$;
- la durée d'une tournée ne doit pas excéder T (temps de travail d'un conducteur) ;
- le transport d'une demande i ne doit pas excéder Δ_i (confort des usagers).

Le problème est donc le suivant : au début de la journée les véhicules sont entreposés au dépôt. Au cours de la journée toutes les demandes doivent être satisfaites. Pour une demande i, un véhicule doit se rendre en son origine o_i pour charger la demande à la date $t(o_i) \in [e_{o_i}, l_{o_i}]$. Le véhicule transporte alors la charge associée à la demande jusqu'à sa destination d_i et la décharge à la date $t(d_i) \in [e_{d_i}, l_{d_i}]$. La durée du transport de la demande i ne doit pas excéder Δ_i : $t(d_i) - t(o_i) \leq \Delta_i$. L'objectif de ce problème est donc de trouver au plus K tournées ainsi que les dates de service $t(x), x \in X$ pour satisfaire toutes les demandes de sorte que la qualité de service soit la meilleure possible.

Notons que la date $t(x)$ correspond à la date de service (chargement ou déchargement) au sommet x. Cette date doit respecter la fenêtre de temps du sommet x : $e_x \leq t(x) \leq l_x$. Le véhicule est autorisé à arriver au sommet x à la date $A(x) \leq t(x)$, on suppose alors qu'il doit attendre à ce sommet $t(x) - A(x)$ unités de temps. Les dates d'arrivée et de service ainsi que la durée d'attente sont illustrées figure 3.19.



δ(x,y) : durée du trajet pour aller de x à y

(e_x, l_x) : fenêtre de temps associée au sommet x

A(x) : date d'arrivée du véhicule en x

t(x) : date du service (chargement ou déchargement) en x

W(x) : temps d'attente en x

FIG. 3.19 – Durées associées aux sommets x et y de X

Remarque 9

Certains auteurs ([CL03] par exemple) incluent des temps de service dans leur modèle. Ceci signifie qu'ils supposent que le service (chargement ou déchargement) en un sommet x dure $\delta_{serv}(x)$ unités de temps. Ils distinguent alors la date de début de service $t(x)$ (qui doit être comprise dans la fenêtre de temps du sommet x) et la date de fin de service $t(x) + \delta_{serv}(x)$. Comme ces temps de service dépendent uniquement du sommet x considéré il est possible de les prendre en compte dans la durée $\delta(x, y)$ en posant $\delta(x, y) = \delta(x, y) + \delta_{serv}(x)$ (on se rend à un sommet x uniquement pour charger ou décharger une demande). On évite ainsi de les manipuler explicitement et donc de surcharger les notations.

Remarque 10

Contrairement aux travaux antérieurs, nous nous focalisons sur les durées (et non sur les

distances) en minimisant une combinaison linéaire des temps d'attente, temps de transport pour l'usager et durée totale de la tournée pour le conducteur.

Ainsi, dans ce modèle, on suppose qu'il est intéressant pour l'opérateur de minimiser le temps de travail des conducteurs (minimisation de la durée totale d'une tournée), tout en assurant une bonne qualité de service (minimisation de la durée de transport d'une demande). De plus, les temps d'attente étant à la fois peu agréables pour l'usager et peu rentables pour l'opérateur, on souhaite également les minimiser.

3.3.2.2 Description d'une solution

Une solution est composée d'au plus K tournées. Une tournée Γ est telle que :

- Γ est une séquence $\{0, 1, \ldots, l+1\}$ de sommets de X qui commence et finit au dépôt ($0 = l + 1 =$ dépôt) ;
- les sommets 1 à l sont tous différents du dépôt ;
- aucun sommet de X autre que le dépôt n'apparaît deux fois dans Γ ;
- si l'origine o_i d'une demande i est dans Γ alors sa destination d_i y est également (et dans cet ordre).

On note x_{succ} le successeur de x dans une tournée Γ et D_Γ l'ensemble des demandes transportées par le véhicule réalisant la tournée Γ.

Une tournée $\Gamma = \{0, 1, \ldots, l+1\}$ doit respecter l'ensemble $\mathcal{C}$ des contraintes suivantes :

- La capacité du véhicule est toujours respectée :

$$\forall x \in \Gamma, \sum_{\substack{y \in \Gamma, \\ 0 \leq y \leq x}} q_y \leq Q \ ;$$

- Les dates de service $t(x)$ des sommets x dans Γ respectent les durées de transport entre les sommets :

$$\forall x \in \Gamma, \ x \neq l+1, \ t(x_{succ}) \geq t(x) + \delta(x, x_{succ});$$

- les durées maximales de transport d'une demande sont respectées :

$$\forall i \in D_\Gamma, \ t(d_i) - t(o_i) \leq \Delta_i;$$

- La durée maximale d'une tournée est respectée :

$$t(l+1) - t(0) \leq T \ ;$$

- Les fenêtres de temps sont respectées :

$$\forall x \in \Gamma, \ t(x) \in [e_x, l_x].$$

Une tournée qui respecte l'ensemble des contraintes $\mathcal{C}$ est dite *valide*.

Pour une tournée $\Gamma = \{0, 1, \ldots, l+1\}$ qui respecte l'ensemble des contraintes $\mathcal{C}$, on peut définir les critères suivants :

– durée totale de la tournée $Dur\acute{e}e(\Gamma)$:

$$Dur\acute{e}e(\Gamma) = t(l+1) - t(0) \ ;$$

– temps de transport $Ride(\Gamma)$ total pour les demandes dans Γ :

$$Ride(\Gamma) = \sum_{i \in D_\Gamma} t(d_i) - t(o_i);$$

– temps d'attente $Wait(\Gamma)$:

$$Wait(\Gamma) = Dur\acute{e}e(\Gamma) - \sum_{\substack{x \in \Gamma, \\ x \neq l+1}} \delta(x, x_{succ}).$$

L'objectif est de minimiser ces trois critères pour chaque tournée. Pour cela, on définit la performance (ou coût) $Perf(\Gamma)$ d'une tournée Γ par la somme pondérée de ces critères :

$$Perf(\Gamma) = \alpha \times Dur\acute{e}e(\Gamma) + \beta \times Ride(\Gamma) + \zeta \times Wait(\Gamma).$$

Les coefficients α, β et ζ sont des réels positifs.

3.3.3 Vérification des contraintes et évaluation d'une tournée

L'algorithme que nous proposons (section 3.3.4) est basé sur une procédure d'insertion. L'insertion d'une demande i dans une tournée Γ est réalisée uniquement si Γ est toujours valide après insertion de i. Rappelons que Γ est valide si elle respecte les contraintes de capacité et les contraintes temporelles.

3.3.3.1 Vérification des contraintes de capacité

On note $charge(x) = \sum_{\substack{y \in \Gamma, \\ 0 \leq y \leq x}} q_y$ la charge du véhicule lorsqu'il part du sommet $x \in \Gamma$. Une tournée Γ respecte la contrainte de capacité si :

$$\forall x \in \Gamma, \ charge(x) \ \leq \ Q \ ;$$

3.3.3.2 Vérification des contraintes temporelles

Pour vérifier facilement les contraintes temporelles, on tient à jour des fenêtres de temps *réduites* $[a_x, b_x]$ pour chaque sommet x de Γ qui sont telles que $t(x)$ doit appartenir à $[a_x, b_x]$ pour que les contraintes temporelles soient respectées.

On détermine les fenêtres $[a_x, b_x]$ pour tout sommet x de Γ grâce à un mécanisme de propagation de contraintes. Dans les règles énoncées ci-après la notation $x \leftarrow y$ signifie que la variable x reçoit la valeur de y.

Les deux premières règles concernent x et le successeur x_{succ} de x, par définition on a $t(x_{succ}) \geq t(x) + \delta(x, x_{succ})$:

Règle $\mathcal{R}_1$: $t(x_{succ})$ ne peut pas commencer avant $a_x + \delta(x, x_{succ})$:

$$a_{x_{succ}} < a_x + \delta(x, x_{succ}) \Rightarrow a_{x_{succ}} \leftarrow a_x + \delta(x, x_{succ})$$

Règle $\mathcal{R}_2$: $t(x)$ ne peut pas commencer après $b_{x_{succ}} - \delta(x, x_{succ})$:

$$b_x > b_{x_{succ}} - \delta(x, x_{succ}) \Rightarrow b_x \leftarrow b_{x_{succ}} - \delta(x, x_{succ})$$

Les deux règles suivantes concernent x et son homologue h dans $\Gamma = \{0, 1, \ldots l+1\}$: si $x = o_i$ alors $h = d_i$, si $x = 0$ alors $h = l+1$. On note Δ la durée maximale autorisée entre $t(x)$ et $t(h)$: $\Delta = \Delta_i$ si x est l'origine de la demande i ; $\Delta = T$ si x représente le dépôt.

Règle $\mathcal{R}_3$: $t(x)$ ne peut pas commencer avant $a_h - \Delta$:

$$a_x < a_h - \Delta \Rightarrow a_x \leftarrow a_h - \Delta$$

Règle $\mathcal{R}_4$: $t(h)$ ne peut pas commencer après $b_x + \Delta$:

$$b_h > b_x + \Delta \Rightarrow b_h \leftarrow b_x + \Delta$$

La cinquième règle concerne le cas où la fenêtre réduite en x devient infaisable.

Règle $\mathcal{R}_5$: b_x ne doit pas devenir plus petit que a_x :

$$b_x < a_x \Rightarrow Echec$$

Algorithme 37 : Propager (calcul des fenêtres réduites)

Entrées : $i \in D$, Γ
Sorties : Reussite

1 $\mathcal{L} \leftarrow \{o_i, d_i\}$;
2 *Reussite* $\leftarrow$ *vrai* ;
3 **Tant que** $\mathcal{L}$ *non vide et Reussite* $=$ *vrai* **faire**
4     $x \leftarrow$ premier élément de $\mathcal{L}$;
5   Supprimer x de $\mathcal{L}$;
6   **Si** x *a un successeur* y **alors**
7   Appliquer les règles $\mathcal{R}_1$ et $\mathcal{R}_2$;
8   **Si** x *a un homologue* h **alors**
9   Appliquer les règles $\mathcal{R}_3$ et $\mathcal{R}_4$;
10   **Si** $b_x < a_x$ *ou* $b_y < a_y$ *ou* $b_h < a_h$ **alors**
11   *Reussite* $\leftarrow$ *faux* ;
12   **Si** *la fenêtre de* x *a changé* **alors** $\mathcal{L} \leftarrow \mathcal{L} \cup x$;
13   **Si** *la fenêtre de* y *a changé* **alors** $\mathcal{L} \leftarrow \mathcal{L} \cup y$;
14   **Si** *la fenêtre de* h *a changé* **alors** $\mathcal{L} \leftarrow \mathcal{L} \cup h$;

Le mécanisme de propagation est décrit par l'algorithme 37. L'algorithme prend en entrée la demande i à partir de laquelle on initialise la propagation.

3.3.3.3 Proposition pour l'évaluation d'une tournée

Pour évaluer le coût de la tournée Γ ($Perf(\Gamma)$), il faut déterminer les dates de service $t(x)$ pour chaque x dans Γ. Les dates $t(x)$ doivent appartenir à la fenêtre de temps réduite $[a_x, b_x]$ pour que les contraintes soient respectées. Dans la fonction objectif, le coût de la tournée Γ est calculé par :

$$Perf(\Gamma) = \alpha \times \mathit{Durée}(\Gamma) + \beta \times \mathit{Ride}(\Gamma) + \zeta \times \mathit{Wait}(\Gamma)$$

Ainsi, chaque date $t(x)$ se voit attribuer naturellement un coefficient λ_x. La valeur de ce coefficient ne dépend que du type de sommet (dépôt de départ, dépôt d'arrivée, origine ou destination d'une demande), on le calcule donc une unique fois de la manière suivante :

$$\lambda_x = \begin{cases} \alpha + \zeta & \text{si} \quad x = l + 1 \\ -\alpha - \zeta & \text{si} \quad x = 0 \\ \beta & \text{si} \quad x \text{ est une destination} \\ -\beta & \text{si} \quad x \text{ est une origine} \end{cases}$$

Il est clair que pour minimiser $Perf(\Gamma)$, les dates de service $t(x)$ doivent être les plus petites possibles pour les sommets x tels que $\lambda_x > 0$ et les plus grandes possibles pour les sommets x tels que $\lambda_x < 0$. On fixe donc les dates $t(x)$ comme décrit par l'algorithme `OptimiserDate` (voir algorithme 38).

Algorithme 38 : OptimiserDate

Entrées : $\Gamma = \{0, 1, \ldots l + 1\}$, //les sommets $x \in \Gamma$ sont associés à la date $t(x)$
Sorties : Γ // la date $t(x)$ des sommets $x \in \Gamma$ a été ajustée au mieux

```
1  stop ← faux ;
2  Tant que stop = faux faire
3      On note s le successeur d'un sommet x et p son prédécesseur ;
4      Soit x tel que
5          λx < 0 et t(x) ≠ min(bx, t(s) − δ(x, s)) ;
6          ou λx > 0 et t(x) ≠ max(ax, t(p) + δ(p, x)) ;
7      Si x n'existe pas alors
8          stop ← vrai ;
9      sinon
10         Si λx < 0 alors
11             t(x) ← min(bx, t(s) − δ(x, s))
12         sinon
13             t(x) ← max(ax, t(p) + δ(p, x))
```

L'évaluation d'une tournée est assurée par l'algorithme `EvaluerTournee` (voir algorithme 39).

Algorithme 39 : EvaluerTournee

Entrées : $\Gamma = \{0, 1, \ldots l+1\}$
Sorties : $Perf(\Gamma)$

1 $x \leftarrow$ premier élément de Γ ;
2 $t(x) \leftarrow b(x)$;
3 1. Initialisation des dates de service **Tant que** $x \neq l+1$ **faire**
4 $\quad y \leftarrow max(a_y, t(x) + \delta(x, y))$;
5 $\quad x \leftarrow y$;
6 2. Optimisation des dates de service **OptimiserDate**(Γ) ;
7 3. Calcul de la fonction objectif $Perf(\Gamma) \leftarrow \alpha \times Durée(\Gamma) + \beta \times Ride(\Gamma) + \zeta \times Wait(\Gamma)$

3.3.4 Proposition d'un algorithme d'insertion pour le DARP

Les sections précédentes fournissent les outils pour vérifier rapidement les contraintes et évaluer le coût d'une tournée. Les procédures décrites précédemment vont permettre de concevoir un algorithme d'insertion pour construire une solution. L'insertion d'une solution est réalisée de la manière suivante. Soit $\Gamma = \{0, \ldots, \gamma_1, \ldots, \gamma_2, \ldots, l+1\}$ une tournée valide. Supposons que l'on souhaite insérer une demande $i \in D$ dans Γ. On choisit alors deux sommets γ_1 et γ_2 dans Γ tels que γ_2 est situé après γ_1. L'origine o_i de la demande est insérée juste après γ_1 et la destination d_i de la demande est insérée juste après γ_2 : Γ devient $\{0, \ldots, \gamma_1, o_i, \ldots, \gamma_2, d_i, \ldots, l+1\}$. Avant d'insérer la demande i, on s'assure de la validité de la tournée après insertion.

3.3.4.1 Validité de la charge après insertion de la demande i

Soit $\Gamma = \{0, \ldots, \gamma, \ldots, \gamma_2, \ldots, l+1\}$. On désigne par $\Gamma_{[\gamma,\gamma_2]}$ la sous-tournée $\{\gamma, \ldots, \gamma_2\}$.

La tournée est valide du point de vue de la charge après insertion de la demande i si elle peut supporter q_i passagers supplémentaires entre γ et γ_2. On vérifie qu'en chaque sommet x de γ à γ_2 la charge augmentée de q_i ne dépasse pas la capacité du véhicule :

$$CN_{charge}(i, \gamma, \gamma_2) \qquad \forall x \in \Gamma_{[\gamma,\gamma_2]},\ charge(x) + q_i \leq Q.$$

3.3.4.2 Conditions nécessaires de validité des contraintes temporelles pour l'insertion de la demande i

Soit Γ la tournée dans laquelle on souhaite insérer la demande i. Les contraintes temporelles sont respectées si, après insertion de i dans Γ, la procédure `Propager`(i, Γ) renvoie *vrai*. Néanmoins, cette procédure peut être lourde et, avant de l'utiliser, on effectue quelques tests simples et rapides. Ces tests sont des conditions nécessaires à la validité de Γ après insertion de la demande i mais pas suffisantes.

Premier test

Soit x un sommet associé à la fenêtre de temps $[e_x, l_x]$ (x représente soit l'origine de la demande i, soit la destination). On souhaite insérer x dans la tournée Γ entre deux som-

mets consécutifs γ et γ'. Les fenêtres de temps réduites de γ et γ' sont $[a_\gamma, b_\gamma]$ et $[a_{\gamma'}, b_{\gamma'}]$ (voir figure 3.20). Une condition nécessaire (CN_1) à la validité de Γ après insertion de x est :

$$CN_1(x,\gamma)\begin{cases} a_\gamma + \delta(\gamma, x) & \leq \; l_x \\ e_x + \delta(x, \gamma') & \leq \; b_{\gamma'} \\ a_\gamma + \delta(\gamma, x) + \delta(x, \gamma') & \leq \; b_{\gamma'} \end{cases}$$

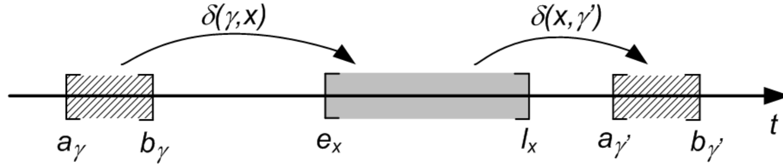


FIG. 3.20 – Fenêtres de temps lors de l'insertion de x entre γ et γ'

Second test

On souhaite insérer la demande i entre deux sommets consécutifs γ et γ' de sorte d'obtenir la sous-tournée $\{\gamma, o_i, d_i, \gamma'\}$ (voir figure 3.21). Une condition nécessaire (CN_2) est :

$$CN_2(i,\gamma)\begin{cases} a_\gamma + \delta(\gamma, o_i) + \delta(o_i, d_i) & \leq \; l_{d_i} \\ a_\gamma + \delta(\gamma, o_i) + \delta(o_i, d_i) + \delta(d_i, \gamma') & \leq \; b_{\gamma'} \\ e_{o_i} + \delta(o_i, d_i) + \delta(d_i, \gamma') & \leq \; b_{\gamma'} \end{cases}$$

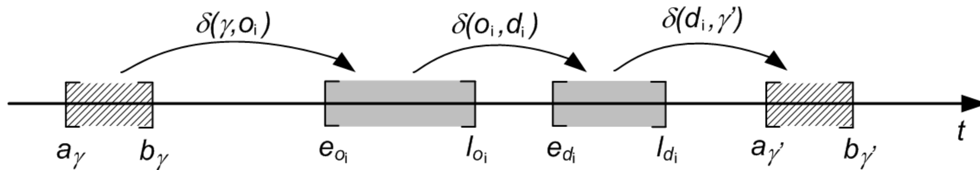


FIG. 3.21 – Fenêtres de temps lors de l'insertion de la demande i entre γ et γ'

Troisième test

On souhaite insérer la demande i dans $\Gamma = \{0, \ldots, \gamma, \gamma' \ldots, \gamma_2, \gamma_2', \ldots, l+1\}$, telle que o_i (origine de i) soit entre γ et γ' et d_i (destination de i) soit entre γ_2 et γ_2'. Une condition nécessaire (CN_3) pour que le temps de transport de la demande i soit respecté est :

$$CN_3(i,\gamma,\gamma_2) \qquad \delta(o_i, \gamma') + \left(\sum_{x \in \Gamma_{[\gamma', \gamma_2[}} \delta(x, x_{succ}) \right) + \delta(\gamma_2, d_i) \leq \Delta_i$$

où $\Gamma_{[\gamma', \gamma_2[}$ est la sous-tournée $\{\gamma', \ldots, (\gamma_2)_{pred}\}$.

La démarche d'insertion d'une demande i dans une tournée Γ est alors la suivante. On teste tout d'abord la validité de la charge puis les conditions nécessaires de validité des

contraintes temporelles. Si elles sont satisfaites on effectue le mécanisme de propagation.

L'insertion d'une demande dans une tournée est réalisée comme décrit par l'algorithme 40.

Algorithme 40 : TestInsertion

Entrées : Γ, $i \in D$ $\gamma \in \Gamma$, $\gamma_2 \in \Gamma$
Sorties : val

```
1  CN ← faux ;
2  Si CN_charge(i, γ, γ2) et CN1(o_i, γ) et CN1(d_i, γ2) sont satisfaites alors
3  |  Si γ ≠ γ2 alors
4  |  |  Si CN3(i, γ, γ2) est satisfaite alors
5  |  |  |  CN ← vrai
6  |  sinon
7  |  |  Si CN2(i, γ) est satisfaite alors
8  |  |  |  CN ← vrai
9  //si les conditions nécessaires sont satisfaites, on calcule l'augmentation de la durée de la
   tournée engendrée par l'insertion de i
10 Si CN = vrai alors
11 |  Γ' ← Γ ;
12 |  (Γ', CS) ← InsérerDemande(Γ', γ, γ2, i) (voir algorithme 41);
13 |  Si CS = vrai alors
14 |  |  Si γ = γ2 alors
15 |  |  |  val ← δ(γ, o_i) + δ(d_i, γ2) + δ(o_i, d_i) − δ(γ, γ2) ;
16 |  |  sinon
17 |  |  |  val ← δ(γ, o_i) + δ(o_i, γ_succ) − δ(γ, γ_succ) + δ(γ2, d_i) + δ(d_i, (γ2)_succ) -
   |  |  |  δ(γ2, (γ2)_succ);
18 Si CN = faux ou CS = faux alors
19 |  val ← −1 ;
```

Algorithme 41 : InsérerDemande

Entrées : Γ, $\gamma \in \Gamma$, $\gamma_2 \in \Gamma$, $i \in D$
Sorties : Γ, *res*

```
1  Insérer le sommet o_i après γ ;
2  Insérer le sommet d_i après γ2 ;
3  res ← Propager(Γ) ;
```

3.3.4.3 Prétraitement sur les fenêtres de temps

L'idée est d'effectuer un prétraitement qui réduit au maximum les fenêtres de temps avant d'exécuter l'algorithme d'insertion. Le fait de réduire préalablement les fenêtres de temps permet d'être plus précis lors du calcul des conditions nécessaires (section 3.3.4.2), l'heuristique est donc plus rapide.

Afin de réduire les fenêtres de temps, on utilise, pour chaque demande, une procédure similaire à la procédure `Propager` avec les règles suivantes :

règle 1 : l'origine o_i d'une demande i ne peut pas être servie avant $\delta(0, o_i)$ et la destination d_i ne peut pas être servie avant $\delta(0, o_i) + \delta(o_i, d_i)$:

$$\begin{aligned} e_{o_i} &= \max(e_{o_i}, \delta(0, o_i)) \\ e_{d_i} &= \max(e_{d_i}, \delta(0, o_i) + \delta(o_i, d_i)) \end{aligned}$$

règle 2 une origine o_i ne peut pas être servie avant $e_{d_i} - \Delta_i$:

$$e_{o_i} = \max(e_{o_i}, e_{d_i} - \Delta_i)$$

règle 3 une origine o_i ne peut pas être servie après $l_{d_i} - \delta(o_i, d_i)$:

$$l_{o_i} = \min(l_{o_i}, l_{d_i} - \delta(o_i, d_i))$$

règle 4 une destination d_i ne peut pas être servie après $l_{o_i} + \Delta_i$:

$$l_{d_i} = \min(l_{d_i}, l_{o_i} + \Delta_i)$$

règle 5 une destination d_i ne peut pas être servie avant $e_{o_i} + \delta(o_i, d_i)$:

$$e_{d_i} = \max(e_{d_i}, e_{o_i} + \delta(o_i, d_i))$$

3.3.4.4 L'algorithme d'insertion

L'algorithme proposé ici repose sur des techniques d'insertion et de propagation de contraintes. Cet algorithme est randomisé afin qu'il puisse générer des solutions diversifiées. Les procédures décrites dans les sections précédentes permettent de développer l'algorithme d'insertion qui construit entièrement une solution. On ajoute les notations suivantes : soit Γ_k la tournée associée au véhicule $k \in [1 \ldots K]$, on note $dep_1(k)$, le premier sommet dans Γ_k (dépôt de départ) et $dep_2(k)$ le dernier sommet dans Γ_k (dépôt d'arrivée).

L'algorithme d'insertion fonctionne alors comme suit : à chaque étape une nouvelle demande est insérée dans une des tournées. Initialement, les tournées Γ_k sont initialisées à $\{dep_1(k), dep_2(k)\}$ pour tout k dans $[1, \ldots K]$. A chaque itération, on tient à jour les éléments suivants :

- l'état des tournées (fenêtres de temps réduites et charge des sommets dans la tournée) ;
- la liste $\mathcal{L}_i$ des insertions possibles pour chaque demande i non encore insérée. Un élément de $\mathcal{L}_i$ est formé de 4 valeurs $(k, \gamma, \gamma_2, val)$, ce qui signifie que la demande i peut être insérée dans la tournée Γ_k, de sorte que, o_i (origine de i) soit placée juste après $\gamma \in \Gamma_k$ et d_i (destination de i) soit placée juste après $\gamma_2 \in \Gamma_k$. La valeur val est le coût associé à cette insertion. Le cardinal $|\mathcal{L}_i|$ de $\mathcal{L}_i$ donne le nombre d'insertions possibles pour la demande i ;
- la liste $\mathcal{LV}_i$ des tournées dans lesquelles il est possible d'insérer une demande i non encore insérée. Le cardinal $|\mathcal{LV}_i|$ de $\mathcal{LV}_i$ donne le nombre d'insertions possibles pour i.

L'algorithme fonctionne alors comme suit :

1. A chaque itération on choisit une demande parmi celles qui sont le plus contraintes de la manière suivante (I_1) :
   - s'il y a des demandes i pour lesquelles $|\mathcal{LV}_i| = 1$, alors on choisit la demande à insérer aléatoirement parmi celles-ci ;
   - sinon on choisit la demande aléatoirement parmi les trois demandes qui possèdent la plus petite valeur $\theta.|\mathcal{LV}_i| + \phi.|\mathcal{L}_i|$ où θ et ϕ sont deux paramètres donnés.
2. Pour la demande i choisie, on considère dans $\mathcal{L}_i$ les N_{el} éléments $(k, \gamma, \gamma_2, val)$ qui possèdent la plus petite valeur val. Pour chacun d'entre eux, on évalue le coût de la tournée Γ_k après insertion de i à la position définie par γ et γ_2. On choisit parmi les trois insertions celle qui engendre la plus petite augmentation du coût de la tournée k et on réalise l'insertion ;
3. On met à jour les listes $\mathcal{LV}_i$ et $\mathcal{L}_i$ pour les demandes i non encore insérées (seuls les éléments $(k, \gamma, \gamma_2, val)$ où k est la tournée qui vient de changer sont susceptibles d'être modifiés).

Algorithme 42 : InsertionGlouton

Entrées :
D //ensemble des demandes
N_{el}, θ, ϕ //paramètres
Sorties :
$\{\Gamma_k\}_{k=1,\ldots,K}$ //ensemble des tournées
$Rejet$ //ensemble des demandes qui n'ont pas été insérées

```
1  // initialisation : les demandes peuvent être insérées dans n'importe quelle tournée :
2  ∀k ∈ [1...K], Γ_k ← {dep_1(k), dep_2(k)} ;
3  ∀k ∈ [1...K], t(dep_1(k)) ← 0,  t(dep_2(k)) ← 0 ;
4  ∀i ∈ D, LV_i ← {1,...,K} ;
5  ∀i ∈ D, ∀k ∈ [1...K] :
6      val_ik ← δ(dep_1(k), o_i) + δ(o_i, d_i) + δ(d_i, dep_2(k)) ;
7      L_i ← {(k, dep_1(k), dep_1(k), val_ik)}_{k=1,...,K} ;
8  Tant que il reste des demandes à insérer faire
9      Choisir la demande i à insérer comme décrit par (I_1) ;
10     Choisir les N_el éléments e = (k, γ, γ_2, val) de L_i qui possèdent la plus petite
       valeur val ;
11     pour chaque élément e faire
12         Γ' ← Insérer(i, Γ_k, γ, γ_2) ;
13         Perf_Γ' ← Evaluer(Γ') ;
14         val_e ← Perf_Γ' - Perf_Γk ;
15     Choisir e_0 aléatoirement parmi les trois éléments e de plus faible val_e ;
16     Soit e = (k, γ, γ_2, val) Γ_k ← Insérer(i, Γ_k, γ, γ_2) ;
17     Pour i non encore insérée faire
18         Mettre à jour L_i et LV_i en utilisant la procédure TesterInsertion ;
19     pour chaque demande i telle que LV_i = ∅ faire
20         Rejet ← Rejet ∪ {i} ;
```

L'algorithme d'insertion est décrit par l'algorithme 42.

Il peut arriver que même après plusieurs exécutions, la procédure `InsertionGlouton` échoue : toutes les demandes n'ont pas été insérées. On tente alors d'améliorer l'efficacité de cet algorithme grâce à deux approches différentes décrites dans les sections suivantes :

- insertion avec apprentissage ;
- insertion avec recherche arborescente.

3.3.4.5 L'algorithme d'insertion avec apprentissage

L'idée est d'identifier et de traiter en priorité les demandes qui posent problème, c'est-à-dire les demandes qui sont souvent difficiles à insérer dans une tournée sans violer les contraintes. Il s'agit probablement (nous revenons sur ce point par la suite) de demandes de transport pour lesquelles les fenêtres de temps sont particulièrement contraintes. Ces demandes de transport sont alors insérées en premier dans les tournées en cours de construction et les demandes « moins contraintes » sont insérées dans un deuxième temps. L'approche que nous proposons se compose donc de deux parties :

- une phase d'apprentissage pendant laquelle les demandes très contraintes sont identifiées ;
- une phase de construction pendant laquelle les demandes identifiées comme « très contraintes » sont insérées en priorité.

Pour cela, on utilise la procédure `InsertionGlouton` et on compte le nombre de fois $n_{rej}(i)$ où une demande est dans $Rejet$ suivant le schéma algorithmique :

1. l'algorithme `InsertionGlouton` est exécuté N_1 fois. Après chaque exécution, on incrémente $n_{rej}(i)$ si la demande i est dans $Rejet$;
2. l'algorithme `InsertionGlouton` est exécuté N_2 fois en insérant prioritairement les demandes i telles que $n_{rej}(i) \neq 0$. Pour cela, on modifie simplement la priorité d'une demande qui devient $\theta.|\mathcal{LV}_i| + \phi.|\mathcal{L}_i| - \Lambda.n_{rej}(i)$.

3.3.4.6 L'algorithme d'insertion avec recherche arborescente

L'idée est de sauvegarder l'état courant des tournées au moment où on fait un choix qui semble déterminant. On crée ainsi des points de sauvegarde. Lorsqu'on se retrouve bloqué (au moins une demande ne peut plus être insérée), on revient au dernier point de sauvegarde et on effectue un choix différent. On parcourt ainsi un arbre de manière heuristique. L'algorithme `InsertionGlouton` est modifié de la façon suivante :

1. Au moment du choix de la demande j à insérer (ligne 9), on crée un point de sauvegarde si $|\mathcal{LV}_j| = 2$ ou $|\mathcal{LV}_j| = 3$ (il n'y a plus que 2 ou 3 tournées possibles pour insérer j). On sauvegarde alors :
   - les tournées courantes ;
   - les listes $\mathcal{L}_i$ et $\mathcal{LV}_i$ pour toute demande i non encore insérée ;
   - la demande choisie j ;
   - la tournée k dans laquelle sera insérée j (cette dernière sauvegarde ne peut se faire qu'après la ligne 15).
2. S'il existe une demande i qui ne peut plus être insérée ($|\mathcal{LV}_i| = 0$), on revient au dernier point de sauvegarde (au lieu de l'ajouter à l'ensemble $Rejet$). On reprend les

tournées sauvegardées, les listes $\mathcal{L}$ et $\mathcal{LV}$ et la demande j à insérer. Les listes $\mathcal{L}_j$ et $\mathcal{LV}_j$ sont modifiées de sorte que la tournée k soit désormais interdite pour l'insertion de j. La demande j est alors insérée dans une nouvelle tournée et si $|\mathcal{LV}_j| = 2$ un nouveau point de sauvegarde est créé.

3. L'algorithme se poursuit jusqu'à ce que toutes les demandes aient été insérées (réussite) ou que tous les points de sauvegarde aient été utilisés (échec).

3.3.5 Ajout des contraintes financières

A partir du modèle et des algorithmes conçus pour le DARP classique, nous construisons un modèle qui prend en compte des contraintes financières. Dans ce modèle le trajet entre deux sommets possède un coût. On considère que l'opérateur ne souhaite pas investir d'argent, donc à chaque instant la quantité totale d'argent disponible (différence entre les revenus, le capital initial et les dépenses) doit être positive. On a une rentrée d'argent pour chaque livraison effectuée.

Ce modèle peut s'appliquer à différentes situations, il peut par exemple représenter une société de taxis : chaque course à un coût (carburant) et le conducteur reçoit le paiement de sa course arrivé à destination. Les rentrées d'argent approvisionnent le compte de la société et les dépenses sont soustraites de ce compte de sorte que la quantité d'argent disponible est commune à toutes les tournées.

On ajoute au modèle initial les hypothèses suivantes :
- pour tous sommets x, $y \in X$, effectuer le trajet de x vers y coûte $c(x, y)$ unités d'argent payable en x,
- la livraison d'une demande i rapporte r_i unités d'argent lorsqu'on arrive à sa destination (c'est-à-dire à la date $t(d_i)$),
- initialement l'opérateur possède R unités d'argent.

On ajoute aux contraintes de capacités et contraintes temporelles la contrainte financière suivante : « à chaque instant la différence entre les revenus (dont l'apport initial) et les dépenses est positive ».

3.3.5.1 Description formelle

On note :

$$\forall x \in X, r_x = \begin{cases} r_i & \text{si } x \text{ est la destination de la demande } i \\ 0 & \text{sinon} \end{cases}$$

$$\forall x \in X, \Gamma.Cash(x) = \sum_{y \in \Gamma_{[0,x]}} r_x - c(y, y_{succ})$$

On note $x_\tau \in \Gamma$ le sommet de plus grand $t(x)$ tel que $t(x) \leq \tau$. On définit la quantité d'argent propre à la tournée Γ à l'instant τ par :

$$\forall \tau \in [0, +\infty[, \Gamma.Cash(\tau) = \Gamma.Cash(x_\tau)$$

On définit la quantité d'argent totale disponible à l'instant τ par :

$$\forall \tau \in [0, +\infty[, Total.Cash(\tau) = R + \sum_{k \in [0,\dots,K]} \Gamma_k.Cash(\tau)$$

Un ensemble de tournées $\{\Gamma_1, \dots, \Gamma_K\}$ est une solution faisable pour la contrainte financière si :

$$\forall \tau \in [0, +\infty[, Total.Cash(\tau) \geq 0$$

Solution du DARPF

Un ensemble de tournées $\{\Gamma_1, \dots, \Gamma_K\}$ est une solution pour le DARP avec contrainte financière (noté DARPF) s'il respecte les contraintes de capacité, les contraintes temporelles et la contrainte financière.

3.3.5.2 Résolution

Définition du problème engendré par les contraintes financières

Dans le DARP classique, la réalisabilité d'une tournée dépend uniquement des sommets qui la composent : les contraintes de capacité et temporelles ne s'appuient que sur les sommets de la tournée considérée.

L'ajout de contraintes financières lie les tournées entre elles puisque la quantité totale d'argent disponible est fonction des dépenses et des revenus engendrés par toutes les tournées. L'insertion d'une demande peut alors remettre en cause la validité de toutes les tournées et plus seulement de la tournée considérée. Pour ne pas perdre la validité des tournées déjà construites une solution consiste à considérer les dates de service dans ces tournées fixées. Ainsi, on suppose, lors de l'insertion d'une demande i dans une tournée Γ, que les dates de service des sommets dans les tournées autre que Γ sont fixées. Il faut alors que l'insertion de i dans Γ soit compatible avec la contrainte financière et les dates de service déjà fixées.

Grâce à cette stratégie, le schéma général de résolution n'est pas modifié. Les contraintes financières sont prises en compte dans la fonction `InsertionDemande` au moment de l'insertion d'une nouvelle demande. Supposons qu'on insère la demande i dans la tournée Γ_{k_0}. Il nous faut distinguer les sommets dont les dates de service sont fixées (tous les sommets des tournées Γ_k, $k \neq k_0$) et les sommets de Γ_{k_0} dont les dates de service ne sont pas fixées (seules les fenêtres de temps réduites sont connues).

Pour cela, on considère la liste $\mathcal{L}_t$ des sommets x de $\bigcup_{\substack{k \in [0,\dots K] \\ k \neq k_0}} \Gamma_k$ triés suivant leur date $t(x)$ croissante. On pose :

$$\mathcal{L}t.Cash(\tau) = R + \textstyle\sum_{\substack{k \in [0,\dots K] \\ k \neq k_0}} \Gamma_k.Cash(\tau)$$

$$\text{et } \mathcal{L}t.Cash(x) = R + \textstyle\sum_{\substack{y \in \mathcal{L}t \\ y avant\ ou\ = x}} r_y - c(y, y_{succ})$$

Ce qui permet de reformuler $Total.Cash(\tau)$:

$$\forall \tau \in [0, +\infty[, Total.Cash(\tau) = \Gamma_{k_0}.Cash(\tau) + \mathcal{L}t.Cash(\tau)$$

Lors de l'insertion d'une demande dans une tournée Γ, on essaie de rendre les quantités $\Gamma.Cash(x)$ et $\mathcal{L}t.Cash(x)$ positives pour tout x en considérant des échanges de ressources (flux financiers) entre les sommets de Γ et ceux de $\mathcal{L}_t$. La figure 3.22 illustre une tournée Γ à deux sommets et une liste $\mathcal{L}t$ à trois sommets. Les quantités $\Gamma.Cash(x)$ et $\mathcal{L}t.Cash(x)$ pour chaque sommet x sont indiquées. Des flux financiers doivent être envisagés entre Γ et $\mathcal{L}t$ pour rendre ces quantités positives en chaque sommet. Ces flux sont indiqués sur les figures 3.23 et 3.24 ainsi que les nouvelles valeur $\Gamma.Cash(x)$ et $\mathcal{L}t.Cash(x)$ qui en résultent pour chaque sommet x.

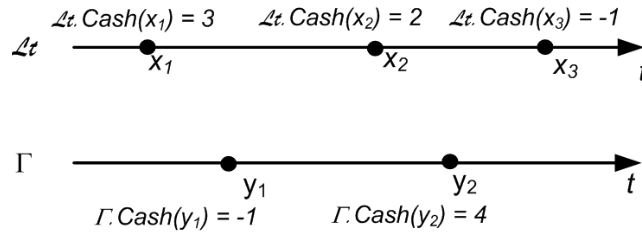


FIG. 3.22 – Etat initial des finances dans $\mathcal{L}t$ et Γ

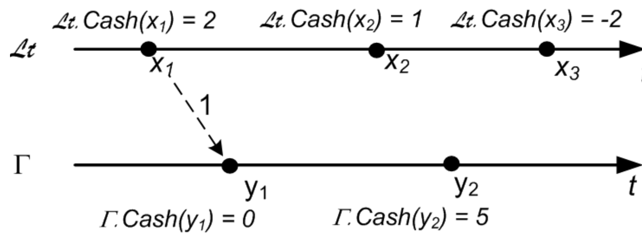


FIG. 3.23 – Un flux financier de x_1 vers y_1 permet d'augmenter $\Gamma.Cash(y_1)$

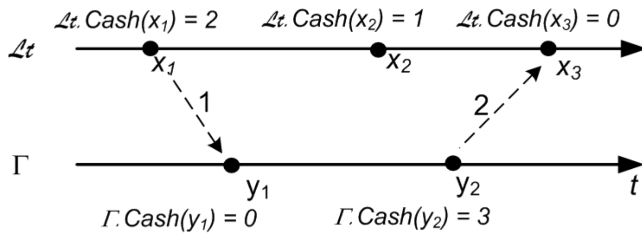


FIG. 3.24 – Etat final des flux financiers dans $\mathcal{L}t$ et Γ

Les flux financiers impliquent de nouvelles contraintes temporelles : s'il y a un flux financier de x vers y alors $t(x) \leq t(y)$. Cette nouvelle contrainte doit bien sûr être prise en compte dans le calcul des fenêtres réduites lors du mécanisme de propagation de contraintes. La fonction `InsérerDemande` est alors modifiée de sorte qu'elle prenne en compte les contraintes financières.

On renomme la fonction `InsérerDemande` en `InsérerDemande_finance`. Cette nouvelle procédure fonctionne de la manière suivante :

1. La demande i est insérée dans Γ_{k_0} ;
2. Les fenêtres de temps réduites dans Γ_{k_0} sont mises à jours avec la procédure `Propager` ;
3. Si les fenêtres de temps réduites sont valides on effectue un premier test qui donne une condition nécessaire pour que les contraintes financières soient valides (voir algorithme `TesterFinance`). Dans ce test, on considère fixes les dates de service des sommets dans les tournées Γ_k, $k \neq k_0$ et on cherche les dates de service des sommets de Γ_{k_0}, de sorte que les sommets qui génèrent plus de revenus que de dépenses soient le plus tôt possible et les autres le plus tard possible. On trie alors l'ensemble des sommets dans l'ordre de leur date de service croissante en calculant pour chaque date τ la différence entre les revenus totaux (y compris le capital initial) et les dépenses engendrées jusqu'en τ. Si pour une date τ, cette différence est négative alors l'algorithme échoue : les contraintes financières ne peuvent pas être respectées ;
4. Si cette condition nécessaire est vérifiée, on applique une procédure récursive `PropagerFinance`. Elle détermine de manière exacte les flux financiers entre $\mathcal{L}t$ et Γ_{k_0} pour qu'en chaque sommet de $\mathcal{L}t$ et de Γ_{k_0} la différence entre revenus et dépenses soit toujours positive (comme le montrent les figures 3.22 à 3.24). Chaque flux financier d'un sommet x vers un sommet y implique une nouvelle contrainte temporelle $t(x) \leq t(y)$. Les dates de service des sommets de $\mathcal{L}t$ étant déjà fixées, la contrainte $t(x) \leq t(y)$ implique une mise à jour des fenêtres de temps réduites de Γ_{k_0}. Dès qu'on fixe un flux financier, on met alors à jour ces fenêtres de temps réduites grâce à la procédure `Propager`. Si la procédure `Propager` échoue cela signifie qu'il n'est pas possible de trouver des flux financiers pour que les contraintes financières soient toujours respectées. Si la procédure réussit, on répète l'opération avec les listes $\mathcal{L}t$ et de Γ_{k_0} qui sont mises à jour de sorte d'intégrer le nouveau flux financier et les nouvelles fenêtres de temps ;
5. Si, à la fin de la procédure `PropagerFinance`, les fenêtres de temps réduites dans Γ sont toujours valides, la procédure réussit (toutes les contraintes ont été testées et sont respectées) sinon elle échoue.

Algorithme 43 : InsérerDemande_finance

Entrées : Γ, $\gamma \in \Gamma$, $\gamma_2 \in \Gamma$, $i \in D$
Sorties : Γ, res

1 Insérer le sommet o_i après γ et le sommet d_i après γ_2 ;
2 **Propager**(Γ) ;
3 **Si** ***TesterFinance*** $=$ *vrai* **alors**
4 $\quad$ $res \leftarrow$ **PropagerFinance**() ;

Algorithme 44 : TesterFinance

Entrées :

$\{\Gamma_k\}_{k=1,\ldots,K}$, //ensemble des tournées

k_0 //numéro de la tournée pour laquelle on est en train de tester l'insertion d'une demande

pour x dans Γ_k, $k \neq k_0$ la date de service $t(x)$ est connue,

pour x dans Γ_{k_0} la fenêtre de temps réduite $[a_x, b_x]$ est connue.

Sorties : *res*

1 //1. on fixe les dates des sommets de Γ_{k_0} de manière favorable aux flux financiers : **pour chaque** $x \in \Gamma_{k_0}$, $x \neq dep_2(k_0)$ **faire**

2 Soit x' le successeur de x dans Γ_{k_0} ;

3 **Si** $r_x - c(x, x') > 0$ //*x génère plus de revenus que de dépenses* **alors**

4 $t(x) \leftarrow a_x$

5 **sinon**

6 $t(x) \leftarrow b_x$

7 //2. s'il existe une date à laquelle les dépenses engagées depuis l'instant 0 ont été plus importantes que les revenus, alors il n'y a pas de solution aux flux financiers :

8 $\mathcal{L} \leftarrow x \in \bigcup_{k \in [1,\ldots K]} \Gamma_k$ dans l'ordre des $t(x)$ croissants ;

9 $i = 1$;

10 $res \leftarrow vrai$;

11 $totalCost \leftarrow R$ //capital initial ;

12 **Tant que** $i < |\mathcal{L}|$ *et* $res = vrai$ **faire**

13 $x \leftarrow \mathcal{L}(i)$ //$i^{ème}$ élément de $\mathcal{L}$;

14 $x' \leftarrow$ successeur de x dans la même tournée que x (si x' n'existe pas, on pose $c(x, x') = 0$) ;

15 $totalCost \leftarrow totalCost + r_x - c(x, x')$;

16 **Si** $totalCost < 0$ **alors**

17 $res \leftarrow faux$;

Algorithme 45 : PropagerFinance

```
    Entrées : ℒ_t, Γ
    Sorties : res
1   Calculer x_0 le premier élément de ℒ_t tel que ℒ_t.Cash(x_0) < 0 ;
2   Calculer y_0 le premier élément de Γ tel que Γ.Cash(y_0) < 0 ;
3   Γ' ← Γ ;
4   Si x_0 et y_0 n'existent pas alors
5       res ← vrai ;
6   sinon
7       CAS 1
8       Si x_0 existe et (y_0 n'existe pas ou t(x_0) < a_{y_0}) alors
9           Calculer y_f le premier élément de Γ qui peut financer x_0 :
10              Γ.Cash(y_f) > −ℒ_t.Cash(x_0) et a_{y_f} ≤ t(x_0);
11          Si y_f n'existe pas alors
12              res ← faux ;
13          sinon
14              pour financer x_0, y_f ne peut pas commencer après t(x_0) :
15              Si b_{y_f} > t(x_0) alors
16                  b_{y_f} = t(x_0);
17                  on recalcule les fenêtres réduites :
18                  p ← Propager(Γ) ;
19                  Si p = vrai alors
20                      Supprimer x_0 de ℒ_t ainsi que tous les sommets qui le précèdent;
21                      Mettre à jour ℒ_t.Cash et Γ.Cash pour prendre en compte le flux
                        financier de y_f vers x_0 ;
22                      res ← PropagerFinance(ℒ_t, Γ') ;
23      CAS 2
24      Si y_0 existe et (x_0 n'existe pas ou t(x_0) > b_{y_0}) alors
25          Calculer x_f le premier élément de ℒ_t qui peut financer y_0 :
26              ℒ_t.Cash(x_f) > −Γ.Cash(y_0) et b_{y_0} ≥ t(x_f);
27          Si x_f n'existe pas alors
28              res ← faux ;
29          sinon
30              y_0 ne peut pas commencer avant t(x_f) :
31              Si a_{y_0} < t(x_f) alors
32                  a_{y_0} < t(x_f);
33                  p ← Propager(Γ) ;
34                  Si p = vrai alors
35                      Supprimer y_0 de Γ' ainsi que tous les sommets qui le précèdent;
36                      Mettre à jour ℒ_t.Cash et Γ.Cash pour prendre en compte le flux
                        financier de x_f vers y_0 ;
37                      res ← PropagerFinance(ℒ_t, Γ') ;
38      Si (y_0 et x_0 existent) et (a_{y_0} ≤ t(x_0) ≤ b_{y_0}) alors
39          Essayer d'appliquer le CAS 1 ;
40          Si on arrive à un échec alors Essayer d'appliquer le CAS 2 ;
```

3.3.6 Expérimentations numériques

Les heuristiques ont été testées sur l'ensemble des instances proposées par Cordeau et Laporte [CL03] (disponibles sur `http://neumann.hec.ca/chairedistributique/data/darp/`). Ils proposent 20 instances comportant de 24 à 144 demandes. Dans chaque instance, chaque demande possède une fenêtre de temps associée, soit à son origine, soit à sa destination. Un temps de service (de durée 10) est imposé pour le chargement et le déchargement. Chaque demande est associée à une charge unitaire. La durée $\delta(x, y)$ pour aller du sommet x au sommet y est égale à la distance $d(x, y)$ entre x et y. La durée maximale d'une tournée est fixée à 480, la durée de transport maximale pour une demande est 90, enfin la capacité du véhicule est égale à 6. Les instances sont divisées en deux groupes : (R1a ... R10a) et (R1b ... R10b). Dans le premier groupe, les fenêtres de temps sont plus serrées. Dans chaque groupe, le nombre de véhicules disponibles pour les instances 1 à 6 est tel que les tournées ne sont pas trop chargées. Le nombre de véhicules disponibles pour les instances 7 à 10 est supposé être le nombre minimal de véhicules nécessaires.

Notons que dans les articles les instances sont nommées R1a ... R10a, R1b, ... R10b tandis que les instances téléchargeables sont nommées pr01 à pr20.

3.3.6.1 Résultats expérimentaux sur le problème de *Dial-a-Ride* classique

Dans cette section, nous présentons tout d'abord une comparaison entre les trois heuristiques que nous proposons pour le *Dial-a-Ride* classique (testées sur PC AMD Opteron, 2.1GHz). Nous comparons ensuite nos résultats à la recherche tabou de Cordeau et Laporte [CL03] (ces auteurs ont effectués leurs tests sur un Pentium 4, 2GHz). Ils proposent une approche différente de la notre puisqu'ils minimisent les distances et non les durées. Cependant, cela permet de voir l'impact de la fonction objectif sur les résultats. Enfin nous proposons une comparaison avec l'algorithme génétique proposé par Jorgensen *et al.*[JLB07] (qui utilisent un PC Intel Celeron, 2GHz). Ces auteurs minimisent une combinaison linéaire de critères qui tend à privilégier le confort des usagers.

3.3.6.1.1 Comparaison des trois heuristiques proposées et efficacité du prétraitement

Les instances de *Dial-a-Ride* proposées par Cordeau et Laporte sont très contraintes à cause, entre autre, des fenêtres de temps. Pour cette raison les heuristiques que nous proposons n'aboutissent pas toujours à une solution. C'est pourquoi, on compare ici l'efficacité des heuristiques en terme de nombre de solutions trouvées sur 100 runs (idéalement un run aboutit à une solution).

Le tableau 3.10 donne le taux de réussite des trois heuristiques sur 100 runs. Les colonnes ont la signification suivante :

- « glouton* »: résultats pour l'heuristique `InsertionGlouton` sans le prétraitement sur les fenêtres de temps ;
- « glouton »: résultats pour l'heuristique `InsertionGlouton` ;
- « apprent. »: résultats pour l'heuristique avec apprentissage (où $N_1 = 5$ et $N_2 = 5$) ;
- « arbre »: résultats pour l'heuristique avec recherche arborescente ;
- nom : nom de l'instance ;

- n : nombre de demandes ;
- K : nombre de véhicules disponibles ;
- réussite : nombre de runs qui aboutissent à une solution ;
- cpu : temps cpu en secondes pour l'ensemble des 100 runs.

instance			« glouton* »		« glouton »		« apprent. »		« arbre »	
nom	n	K	réussite	cpu (s)	réussite	cpu (s)	réussite	cpu (s)	réussite	cpu (s)
R1a	24	3	100	1.25	100	0.29	100	0.29	100	0.42
R2a	48	5	99	4.08	97	1.08	100	1.08	100	1.29
R3a	72	7	96	7.21	93	1.96	100	2.01	100	2.52
R4a	96	9	100	7.00	100	3.94	100	3.75	100	4.24
R5a	120	11	98	17.57	94	6.10	100	6.25	100	6.79
R6a	144	13	95	22.51	99	7.97	100	7.97	100	9.41
R7a	36	4	97	3.39	90	0.56	100	0.60	100	0.71
R8a	72	6	33	8.07	42	2.12	100	4.62	100	293.23
R9a	108	8	0	8.00	1	3.00	51	33.44	30	441.02
R10a	144	10	0	23.09	0	8.01	1	76.02	25	25371.00
R1b	24	3	100	1.47	100	0.60	100	0.58	100	0.82
R2b	48	5	100	5.63	99	1.87	100	1.79	100	1.98
R3b	72	7	90	9.24	91	3.14	100	3.31	100	3.45
R4b	96	9	95	16.12	94	6.43	100	6.50	100	6.60
R5b	120	11	100	16.00	100	10.19	100	9.72	100	10.09
R6b	144	13	99	26.63	99	12.85	100	12.39	100	13.56
R7b	36	4	93	1.99	92	1.09	100	1.12	100	1.23
R8b	72	6	92	6.65	84	3.74	100	4.20	100	4.54
R9b	108	8	22	14.92	28	8.69	99	29.85	99	1215.92
R10b	144	10	2	22.63	13	12.68	90	71.18	45	1345.80
moyenne			75.55	11.17	75.80	4.82	92.05	13.83	89.95	1436.73

TAB. 3.10 – Résultats donnés par les trois heuristiques pour 100 runs (* signifie que l'heuristique a été exécutée sans la réduction préalable des fenêtres temps)

On constate que le calcul préalable sur les fenêtres de temps réduit de 57% le temps d'exécution de l'heuristique `InsertionGlouton`. Ce prétraitement est donc particulièrement utile dans le cadre de notre heuristique avec propagation de contraintes.

On remarque que l'heuristique `InsertionGlouton` ne parvient pas à trouver une solution à chaque run. Néanmoins, sur les 100 runs, elle donne une solution pour toutes les instances sauf la R10a en des temps de calcul très courts (4,82 secondes en moyenne). Elle donne, en moyenne, 75.80 % de réussite. L'ajout d'une phase d'apprentissage augmente considérablement ce pourcentage qui passe à 92.05 %. On constate, cependant, que l'instance R10a pose toujours des difficultés puisqu'elle n'est résolue qu'une fois sur 100 runs. L'heuristique avec recherche arborescente est beaucoup plus efficace sur cette instance (25% de résussite). Son temps d'exécution a été limité à 300 secondes. Cependant, elle a un taux de réussite moyen plus faible que l'heuristique avec apprentissage et des temps de calcul beaucoup plus élevés.

3.3.6.1.2 Comparaison avec les résultats de [CL03]

Le tableau 3.11 compare les résultats de l'heuristique avec apprentissage (meilleur ré-

sultat sur 100 runs) à la recherche tabou de Cordeau et Laporte [CL03]. La comparaison entre les deux méthodes est difficile car [CL03] minimisent les distances parcourues alors que notre méthode minimise une combinaison linéaire des différentes durées. Cependant, minimiser les distances a pour effet de minimiser également la durée totale des tournées. Pour effectuer la comparaison des deux méthodes, nous avons donc choisi d'utiliser les coefficients $\alpha = 10$, $\beta = 1$ et $\zeta = 1$ afin de donner plus de poids à la durée totale des tournées.

Nous reportons dans le tableau les valeurs suivantes :

- Durée = $\sum_{k \in [1...K]} Durée(\Gamma_k)$;
- Wait = $\sum_{k \in [1...K]} Wait(\Gamma_k)$;
- Ride = $\sum_{k \in [1...K]} Ride(\Gamma_k)$;
- Distance : distance parcourue par l'ensemble des véhicules ;
- cpu : temps total en secondes.

Dans [CL03], les meilleurs résultats sont obtenus avec 10^5 itérations mais les temps de calcul sont donnés seulement pour 10^4 itérations. En supposant qu'une itération a un temps constant, on a multiplié par 10 le temps donné pour 10^4 itérations. Pour notre méthode, le temps donné est celui nécessaire à l'ensemble des 100 runs.

	heuristique avec apprentissage cette thèse					recherche tabou Cordeau et Laporte [CL03]				
instance	Durée	Wait	Ride	Distance	cpu(s)	Durée	Wait	Ride	Distance	cpu(s)
R1a	840.3	60.4	653.6	299.9	0.3	881.2	211.2	1095.0	190.0	1140
R2a	1622.1	138.3	1860.2	523.8	1.1	1985.9	723.9	1976.7	302.1	4836
R3a	2593.0	135.5	2073.0	1017.5	2.0	2579.4	607.3	3586.7	532.1	10308
R4a	3414.3	241.5	2872.9	1252.9	3.8	3583.2	1090.4	5021.3	572.8	17262
R5a	3824.3	115.6	3900.3	1308.8	6.3	3870.0	833.0	6156.5	637.0	27744
R6a	4632.0	142.5	4369.2	1609.6	8.0	5056.8	1375.4	7273.5	801.4	32322
R7a	1221.6	30.3	1054.5	471.3	0.6	1452.1	440.3	1508.9	291.7	2634
R8a	2477.9	85.3	2236.9	952.6	4.6	2345.2	410.3	3691.5	494.9	12264
R9a	3470.2	25.8	3065.6	1284.4	33.4	3155.5	323.1	5621.8	672.4	30306
R10a	4673.0	118.8	5152.7	1674.1	76.0	4480.1	721.3	7163.7	878.8	52518
R1b	783.3	10.9	664.7	292.4	0.6	965.1	320.6	1041.5	164.5	1158
R2b	1480.9	25.3	1351.8	495.6	1.8	1564.7	308.7	2393.8	296.1	4974
R3b	2367.4	1.5	2110.6	925.9	3.3	2263.7	330.4	3788.8	493.3	11124
R4b	2995.9	18.0	2598.6	1057.9	6.5	2882.2	426.3	4632.8	535.9	18708
R5b	3573.1	8.1	3142.8	1164.9	9.7	3595.6	605.9	6104.7	589.7	32598
R6b	4312.7	8.7	4116.7	1424.0	12.4	4072.5	448.9	7347.4	743.6	44220
R7b	1229.8	58.6	1007.4	451.2	1.1	1097.3	129.0	1762.0	248.2	2538
R8b	2452.0	52.5	2032.7	959.5	4.2	2446.5	523.4	3659.0	462.7	13716
R9b	3574.1	85.3	3335.0	1328.8	29.9	3249.3	487.3	5581.0	602.0	30768
R10b	4659.0	118.2	3839.4	1660.8	71.2	4041.0	362.4	7072.3	798.6	55446
moyenne	2809.8	74.1	2571.9	1007.8	13.8	2778.4	534.0	4323.9	515.4	20329

Tab. 3.11 – Comparaison des résultats entre notre heuristique qui minimise les durées et celle de Cordeau et Laporte qui minimise les distances

Les résultats donnés par le tableau 3.11 montrent, comme on peut s'y attendre, que la fonction objectif détermine complètement les caractéristiques des tournées. L'heuristique de [CL03] donne des distances deux fois plus faibles en moyenne que notre heuristique. Notre heuristique donne des temps de transport (*Ride*) environ deux fois plus faibles en moyenne et des temps d'attente environ 7 fois plus faibles. Les durées des tournées sont comparables. Il est clair que l'heuristique de [CL03] est plus favorable à l'opérateur alors que notre heuristique privilégie le confort des usagers.

On remarque également que notre heuristique est très rapide (seulement 13.8 secondes en moyenne pour l'ensemble des 100 runs contre 20329 secondes en moyenne pour celle de [CL03]). Cette différence importante s'explique par le fait que [CL03] met en jeu une recherche locale assez sophistiquée alors nous utilisons une simple heuristique de construction. Encore une fois, il est difficile de comparer de manière juste ces temps de calcul, la fonction objectif étant différente d'une méthode à l'autre.

3.3.6.1.3 Comparaison avec les résultats de [JLB07]

Le tableau 3.12 compare les résultats de l'heuristique avec apprentissage (meilleur résultat sur 100 runs) avec les résultats de l'algorithme génétique proposé par Jorgensen *et al.*[JLB07] qui minimisent une combinaison linéaire des critères suivants :

(1) distance totale ;
(2) temps de transport supplémentaire par rapport au temps de transport minimal pour chaque demande ;
(3) temps d'attente des passagers (temps d'attente en chaque sommet multiplié par le nombre de passagers dans le véhicule) ;
(4) durée totale des tournées ;
(5) pénalité sur la violation des fenêtres de temps ;
(6) pénalité sur la violation du temps de transport maximal des demandes ;
(7) pénalité sur la violation de la durée maximale des tournées.

Les coefficients w_i des critères i, $i = 1 \dots 7$ sont fixés à : $w_1 = 8$, $w_2 = 3$, $w_3 = 1$, $w_4 = 1$, $w_5 = n$, $w_6 = n$ et $w_7 = n$. D'après les auteurs, ces coefficients ont été choisis pour minimiser prioritairement le temps de transport total des passagers. Le but de leur méthode étant de donner la priorité au confort des usagers.

Remarquons qu'ils relaxent certaines contraintes. Ainsi la fonction objectif inclut une pénalité pour prendre en compte les contraintes violées. Ils n'ont pas de garantie que le processus s'arrête sur une solution du problème (contrairement à notre méthode et à celle de Cordeau et Laporte). Ils fournissent la durée totale, le temps de transport total et le temps d'attente total pour 5 runs de leur algorithme génétique mais pas la distance totale ni le coût correspondant aux contraintes violées. Dans le tableau 3.12, nous reportons le meilleur résultat sur les 5 runs.

Pour effectuer la comparaison avec notre méthode, nous avons donc choisi d'utiliser les coefficients $\alpha = 1$, $\beta = 8$ et $\zeta = 1$. Les champs sont les mêmes que dans le tableau 3.11, sauf la distance qui a été remplacée par la somme des trois critères. [JLB07] ne donnent des résultats que pour 13 instances parmi les 20. La moyenne concerne donc uniquement ces 13 instances.

	heuristique apprent. cette thèse					algorithme génétique Jorgensen *et al.*[JLB07]				
instance	Durée	Wait	Ride	Somme	cpu(s)	Durée	Wait	Ride	Somme	cpu(s)
R1a	922.4	128.7	301.6	**1352.6**	0.3	1039	260	310	1609	334.2
R2a	1870.3	392.5	749.7	**3012.5**	1.3	1994	514	1330	3838	685.8
R3a	2719.6	205.0	860.2	**3784.8**	2.8	2781	301	2894	5976	1294.8
R4a	3300.1	165.4	1112.2	4577.7	5.3	-	-	-	-	-
R5a	4009.1	253.1	1106.6	**5368.8**	8.6	4274	527	4837	9638	3493.8
R6a	4941.3	200.8	1550.6	6692.8	12.0	-	-	-	-	-
R7a	1274.1	85.4	465.5	1825.1	0.7	-	-	-	-	-
R8a	2462.9	99.5	1171.7	3734.1	9.0	-	-	-	-	-
R9a	3588.4	40.8	2758.0	**6387.2**	51.6	3526	32	6719	10277	2446.8
R10a	4633.5	57.7	3162.4	**7853.5**	105.4	5025	246	8341	13612	3958.8
R1b	845.2	66.4	234.6	**1146.2**	0.7	928	164	549	1641	327.6
R2b	1567.8	17.6	323.9	**1909.2**	2.4	1710	162	1300	3172	703.2
R3b	2596.2	18.5	685.0	3299.6	4.8	-	-	-	-	-
R4b	3151.2	11.4	913.1	4075.6	9.0	-	-	-	-	-
R5b	3812.4	41.3	940.9	**4794.7**	13.2	4336	568	4720	9624	3535.8
R6b	4689.6	22.2	1097.5	**5809.3**	16.9	5227	513	6397	12137	4873.8
R7b	1235.0	22.3	343.9	**1601.2**	1.5	1316	128	784	2228	497.4
R8b	2473.5	54.4	860.1	3388.1	5.2	-	-	-	-	-
R9b	3682.0	84.8	1192.4	**4959.2**	31.8	3676	177	5358	9211	2679.6
R10b	4773.5	50.5	1861.4	**6685.3**	133.8	4678	85	8119	12882	3984.6
moyenne	2949.9	106.4	1148.7	4205.0	28.5	3116.2	282.8	3973.7	7372	2216.6

TAB. 3.12 – Comparaison des résultats entre notre heuristique et celle de Jorgensen *et al.* (la moyenne concerne uniquement les 13 instances pour lesquelles les auteurs ont fourni un résultat)

Le tableau 3.12 montre l'efficacité de notre heuristique pour minimiser les critères correspondant au confort du client : les temps de transport et les temps d'attente sont plus faibles pour 12 instances sur 13. En moyenne, notre heuristique fournit des temps de transport 71% plus faibles et des temps d'attente 62% plus faibles pour des durées totales des tournées équivalentes. Les résultats confirment également la rapidité de notre méthode.

Récemment, Parragh *et al.* ([PDH10]) ont publié de très bons résultats. Ils utilisent une VNS avec la même fonction objectif que Cordeau et Laporte. Ils proposent une adaptation de leur algorithme pour prendre en compte la fonction objectif de Jorgensen *et al.*. Cependant, ils ne fournissent pas les valeurs des temps d'attente ni les temps de transport pour les usagers.

3.3.6.2 Résultats expérimentaux sur le problème de *Dial-a-Ride* avec contraintes financières

Pour tester l'heuristique avec flux financiers on a modifié les instances de Cordeau de la manière suivante :

- chaque destination est associée à un revenu ;
- chaque couple de sommets (x, y) est associé à un coût financier qui représente la somme à payer pour aller de x à y ;
- les temps de service sont supprimés ;

– la durée maximale de transport d'une demande est fixée à 100.

Les instances sont renommées F1a à F10b.

Avant d'exécuter les heuristiques, les fenêtres sont réduites en fonction des contraintes du problème, comme pour le DARP classique (voir section 3.3.6.1).

La stratégie de l'heuristique avec recherche arborescente est légèrement différente. En effet, le processus d'insertion dans le DARP avec contraintes financières présente deux différences majeures :

– même si une demande peut être insérée dans plusieurs véhicules, du point de vue des contraintes temporelles, il se peut qu'elle soit trop coûteuse pour être réellement insérée ;
– l'insertion d'une nouvelle demande qui génère un revenu important peut rendre insérables des demandes qui ne l'étaient pas.

Pour prendre en compte ces différences on modifie la création et l'utilisation des points de sauvegarde comme suit :

– on crée un point de sauvegarde à l'insertion de chaque demande ;
– à la fin du processus, si plus aucune demande ne peut être insérée on utilise les points de sauvegarde ;
– l'insertion d'une demande ne peut créer qu'un point de sauvegarde (arbre binaire) ;
– une exécution est limitée à 600 secondes.

Le tableau 3.13 compare les trois heuristiques en terme de temps de calcul et d'efficacité (nombre de fois où l'heuristique fournit une solution). De même que pour le tableau 3.10 les colonnes ont la signification suivante :

– « glouton »: résultats pour l'heuristique `InsertionGlouton` ;
– « apprent. »: résultats pour l'heuristique avec apprentissage (où $N_1 = 5$ et $N_2 = 5$) ;
– « arbre »: résultats pour l'heuristique avec recherche arborescente ;
– nom : nom de l'instance ;
– réussite : nombre de runs (sur 100) qui aboutissent à une solution ;
– cpu : temps cpu en secondes pour l'ensemble des 100 runs.

On constate que l'ajout des contraintes financières augmente considérablement les temps de calcul (89.1 secondes en moyenne pour l'heuristique basique contre 4.8 dans le DARP classique). On constate également que l'efficacité relative des trois heuristiques est la même que dans le cas classique : l'heuristique avec apprentissage fournit les meilleurs résultats.

Le tableau 3.14 donne la durée totale des tournées, le waiting time et le riding time obtenus avec l'heuristique apprentissage.

	« glouton »		« apprent. »		« arbre »	
instance	réussite	cpu (s)	réussite	cpu (s)	réussite	cpu (s)
F1a	49	2.31	100	4.68	95	14.7
F2a	79	9.19	100	11.94	100	15.3
F3a	6	25.49	79	160.62	79	504.5
F4a	16	65.13	88	321.91	97	1431.3
F5a	54	118.49	100	187.73	100	889.8
F6a	80	99.24	100	120.9	100	665.4
F7a	3	6.05	99	38.79	44	99.5
F8a	72	19.52	100	27.03	100	62.0
F9a	69	49.34	100	73.89	100	394.0
F10a	40	113.12	99	311.43	100	2357.8
F1b	98	3.62	100	3.7	100	4.2
F2b	83	25.59	100	33.78	100	46.3
F3b	29	43.73	100	147.33	100	409.4
F4b	25	112.19	100	415.85	100	1848.4
F5b	17	275.97	79	1503.67	93	4514.5
F6b	9	332.45	77	2135.61	92	9995.2
F7b	48	11.73	100	24.91	100	28.8
F8b	34	53.7	100	130.43	100	311.5
F9b	31	178.39	99	550.74	100	1383.4
F10b	50	237.37	99	507.43	100	2706.1
moyenne	44.6	89.131	95.95	335.6185	95.0	1384.1

TAB. 3.13 – Comparaison des trois heuristiques pour le DARP avec contraintes financières

Instance	Durée	Wait	Ride	cpu(s)
F1a	392.7	115.3	283.7	4.7
F2a	592.8	98.1	970.6	11.9
F3a	1209.0	181.9	1130.0	160.6
F4a	1236.8	135.4	1231.5	321.9
F5a	1305.1	63.6	1568.7	187.7
F6a	1567.6	97.4	2386.4	120.9
F7a	562.1	102.4	666.8	38.8
F8a	1104.1	142.2	1128.2	27.0
F9a	1190.9	30.8	2192.2	73.9
F10a	1564.8	26.0	2762.9	311.4
F1b	377.2	112.7	369.1	3.7
F2b	457.9	1.6	711.9	33.8
F3b	867.4	9.6	1181.1	147.3
F4b	1029.6	57.6	1254.4	415.9
F5b	1122.0	47.4	2074.6	1503.7
F6b	1385.0	6.6	2272.8	2135.6
F7b	459.6	46.8	850.3	24.9
F8b	876.6	63.9	1129.1	130.4
F9b	1101.7	11.7	2011.8	550.7
F10b	1485.6	39.9	2551.7	507.4
moyenne	994.4	69.5	1436.4	335.6

TAB. 3.14 – Caractéristiques des solutions

3.3.7 Conclusion

Cette section présente trois heuristiques basées sur des techniques d'insertion et de propagation de contraintes pour le problème de *Dial-a-Ride* (DARP). Le but de ces méthodes est, d'une part, de fournir rapidement des solutions qui privilégient le confort des usagers (temps d'attente et temps de transport minimaux) et, d'autre part, de pouvoir s'adapter à une extension du DARP avec contraintes financières. Les trois heuristiques ont été testées sur les instances de Cordeau et Laporte [CL03] et les résultats ont été comparés à ceux de la littérature. La comparaison entre les différentes méthodes est difficile car d'un article à l'autre les auteurs ne minimisent pas les mêmes critères. Cependant, ces expérimentations ont montré que nos heuristiques sont capables de fournir des résultats de bonne qualité dans des temps de calcul très courts.

Les trois heuristiques ont été adaptées pour prendre en compte des contraintes financières. Les extensions de type flux financiers nous semblent particulièrement intéressantes car, d'une part, elles correspondent à des situations nouvelles qui sont d'actualité avec l'émergence du transport à la demande et des problèmes de tarification induits. D'autre part, elles obligent à redéfinir de nouvelles méthodes car ces extensions contraignent fortement le problème de départ rendant difficile l'adaptation des méthodes existantes.

3.4 Perspectives : transport à la demande avec correspondances

Dans la section précédente, nous nous sommes intéressés à un problème de transport à la demande avec des contraintes financières. Dans le but de proposer des problèmes proches de situations réels, nous envisageons également de traiter le problème des correspondances dans le transport à la demande. Nous présentons ici brièvement le problème et la modélisation envisagée.

3.4.1 Présentation du problème

Les clients de transports en commun savent qu'il est courant d'avoir à prendre des correspondances : RER / métro, bus / bus, train / navette ... Dans le cadre du transport à la demande, les correspondances peuvent être à la fois avantageuses pour l'opérateur et pour l'usager. En effet, elles permettent d'éviter certains détours et réduisent donc la longueur des tournées. Cela a pour effet de diminuer les coûts pour l'opérateur et de raccourcir la durée du trajet pour l'usager.

Un exemple est donné figure 3.25. Une demande i est repérée par son origine o_i et sa destination d_i. La figure 3.25 (a) illustre deux tournées sans correspondance. La première (en trait plein) transporte les demandes 1, 2, 3 et 6. La seconde (en trait pointillé) transporte les demandes 4 et 5. La demande 1 est donc transportée dans la première tournée. La figure 3.25 (b) montre ces mêmes tournées avec une correspondance pour la demande 1. On constate que la longueur de la première tournée (en trait plein) diminue significativement tandis que la longueur de la deuxième tournée augmente très peu. De plus, le détour effectuer par la demande 1 est plus petit.

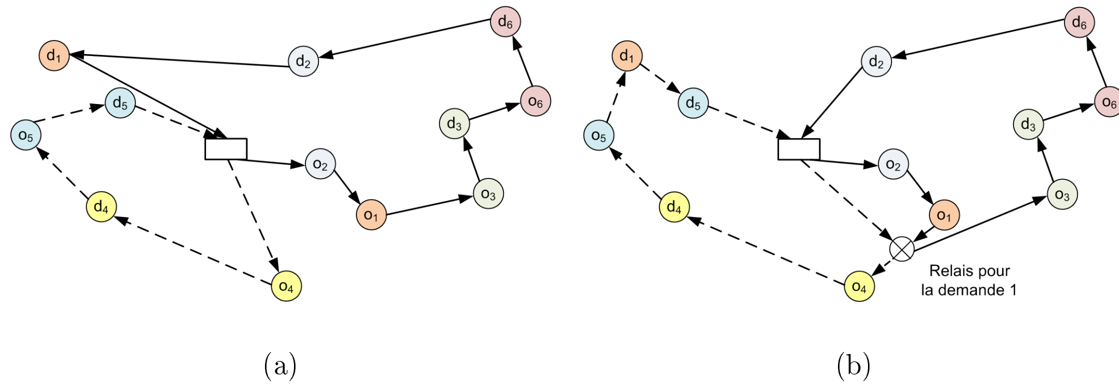


FIG. 3.25 – (a) sans correspondance ; (b) avec correspondance

Nous envisageons uniquement le cas dans lequel un usager prend au plus une correspondance. Cela signifie qu'il effectue soit un trajet direct (un seul véhicule l'emmène de son origine vers sa destination), soit un trajet avec une correspondance (un premier véhicule l'emmène de son origine vers un point relais, puis un second véhicule l'emmène du point relais à sa destination). Il est clair que la correspondance est intéressante pour l'usager s'il n'attend pas trop longtemps au point relais et qu'elle lui permet de gagner du temps. C'est dans cet optique que nous envisageons de traiter le problème.

3.4.2 Modélisation et résolution envisagées

Nous proposons de créer des points relais de manière dynamique. Nous supposons qu'un point relais peut se situer n'importe où, ses coordonnées sont calculées de manière à minimiser le détour effectué par les véhicules concernés. Par exemple, sur la figure 3.25 (a) on constate qu'il serait intéressant d'avoir un point relais entre le sommet origine o_1 et le sommet origine o_4 pour acheminer la demande 1. On crée donc le point relais au milieu de ces deux sommets. En pratique, cela suppose qu'un point relais peut être dans n'importe quelle rue ou ville.

Dans les modèles et algorithmes, la présence du point relais se traduit principalement de deux manières :

- une contrainte temporelle supplémentaire est à prendre en compte dans la propagation de contraintes : le véhicule qui dépose le client au point relais doit y passer avant le véhicule qui récupère le client à ce point ;
- les relais lient les tournées entre-elles. Le mécanisme de propagation agit donc sur l'ensemble des tournées : l'origine de la demande qui vient d'être insérée est le point de départ de la propagation. Les sommets affectés par cette insertion sont listés et traités tour à tour. Ainsi, contrairement au DARP classique, les sommets qui peuvent être affectés par l'insertion ne se limitent pas aux sommets de la tournée courante.

3.5 Conclusion

Dans ce chapitre nous avons étudié deux problèmes de ramassage et livraison (*pickup and delivery* problem).

Tout d'abord nous nous sommes intéressé au problème de la grue, plus connu sous le nom de *Stacker Crane Problem* (SCP). Nous avons choisi de l'étudier sous sa forme préemptive et asymétrique. Le SCP préemptif et asymétrique consiste à déterminer la tournée d'un unique véhicule de capacité unitaire pour satisfaire un ensemble de demandes. Une demande est définie par une origine, une destination et une charge qui doit être transportée de l'origine vers la destination. L'hypothèse de préemption autorise le véhicule à déposer temporairement une charge pour traiter une autre demande. Nous avons montré qu'il est possible de représenter les solutions de ce problème comme un arbre *biparti ordonné*. Nous avons alors pu reformuler le SCP comme un problème original de construction d'arbre. A notre connaissance, il s'agit de la première approche de ce type pour des problèmes de *pickup and delivery*. Nous avons testé l'efficacité de cette méthode sur des instances de la TSPLIB que nous avons généralisées pour ce problème spécifique. Nous avons, de plus, utilisé un algorithme exact de *branch and cut* pour obtenir les solutions optimales. Les expérimentations numériques montre que la méthode s'approche de la solution optimale dans des temps de calcul très court.

Ensuite, nous avons considéré un problème de transport à la demande avec contraintes financières. Les problèmes de transport à la demande sont connus sous le nom de *Dial-a-Ride Problem*. Ils consistent à déterminer les tournées réalisées par une flotte de véhicules pour satisfaire un ensemble de demandes de transport. Ce problème est différent de la majorité des problèmes de *pickup and delivery* car il s'agit de transporter des personnes, la qualité de service tient donc une place prépondérante. Nous avons fourni pour ce problème une méthode d'insertion basée sur des techniques de propagation de contraintes. La méthode a tout d'abord été testée sur des instances pour le *Dial-a-Ride Problem* classique (on ignore les contraintes financières) afin d'être comparée aux résultats de la littérature. Même si la comparaison est difficile à cause des objectifs qui sont différents, notre méthode donne, pour l'objectif que nous considérons, de bons résultats. De plus, les temps de calcul sont beaucoup plus faibles que ceux des méthodes existantes. A notre connaissance, le problème avec contraintes financières n'a jamais été étudié. Néanmoins, ce problème nous paraît intéressant car il met en avant de nouvelles problématiques de résolution et peut s'inscrire dans le cadre d'études de problèmes plus réalistes.

Chapitre 4

Problèmes de tournées et d'ordonnancement

Dans ce chapitre, nous nous intéressons à un problème étudié depuis peu de tournées de véhicules sur nœuds qui inclut un problème d'ordonnancement de type placement. Le problème traité est le 2L-CVRP (*two-dimensional Loading Capacitated Vehicle Routing Problem*) dans lequel des colis en deux dimensions doivent être livrés à des clients. Pour cela une flotte de véhicules homogènes et de capacité limitée est disponible. Afin de situer ce problème parmi les problèmes classiques de tournées de véhicules sur nœuds (souvent désignés par l'acronyme VRP pour *Vehicle Routing Problem*) nous commençons par donner une brève description des ceux-ci. Nous décrivons ensuite la méthode que nous proposons pour résoudre le 2L-CVRP. Notre méthode repose sur une relaxation originale du problème. Elle reprend des idées déjà introduites dans le chapitre 2 concernant la résolution du problème de placement.

4.1 Présentation générale des problèmes de Tournées de Véhicules (*Vehicle Routing Problem* - VRP)

Dans les problèmes de tournées de véhicules (*Vehicle Routing Problem* - VRP), une flotte de véhicules doit fournir un ensemble de clients dont les demandes sont connues a priori. Le but est de trouver des tournées de coût minimal permettant de livrer tous les clients et respectant les contraintes de capacité des véhicules. Contrairement aux problèmes de *pickup and delivery* dans lesquels les clients peuvent être collecteurs et fournisseurs, dans les problèmes de type VRP les clients sont uniquement collecteurs, ils reçoivent donc leur marchandise du dépôt (pour faire le parallèle avec la classification des problèmes de *pickup and delivery* que nous proposons dans le chapitre 3, nous pourrions nommer ce cas « *one-to-many* », et l'insérer dans les problèmes d'échanges entre dépôt et clients). Cependant, deux variantes du VRP incluent la possibilité d'avoir des clients fournisseurs, il s'agit du VRPPD (VRP *with Pickups and Deliveries*) et du VRPB (VRP *with Backhauls*).

Nous présentons dans cette section différents problèmes classiques de VRP. Nous verrons que les dénominations des problèmes sont parfois ambigües car tous les auteurs ne les interprètent pas de la même façon. En particulier, l'acronyme VRP désigne, suivant

les cas, soit la classe des problèmes de tournées sur nœud soit un problème de tournées particulier. Les définitions que nous proposons s'appuient sur des publications et ouvrages récents ([CLSV07], [GILM08a], [Pri09b], [PCF08][AFGS10], [TV02], [BBV08]) d'auteurs connus du domaine. Nous n'évoquons que des problèmes statiques : les demandes des clients sont connues a priori. Nous signalons les différences rencontrées dans la définition du VRP parmi ces publications. Les problèmes de tournées de véhicules sur nœuds les plus connus sont les suivants :

Vehicle Routing Problem (VRP) Comme il a été mentionné précédemment, il existe des définitions multiples du VRP. Nous donnons tout d'abord une définition générale, qui s'appuie sur [TV02] :
« Le VRP est un problème concernant la distribution de biens entre dépôt(s) et clients. Une solution du VRP est un ensemble de tournées, effectuées chacune par un véhicule qui commence et finit sa tournée à son dépôt. Le but est de livrer tous les clients en minimisant le coût total de transport et en respectant les contraintes opérationnelles. »
Cette définition est très générale car les auteurs considèrent le VRP comme une famille de problèmes et non comme un problème particulier.

En revanche, [CLSV07] donne une définition précise du VRP dans le cas symétrique :
« Le VRP symétrique est défini par un graphe complet non orienté $G = (V, E)$. L'ensemble $V = \{0, \ldots, n\}$ est l'ensemble des sommets. Chaque sommet $i \in V - \{0\}$ représente un client, le sommet 0 représente le dépôt. A chaque arc $e \in E$ est associé un coût de transport c_e. Une flotte donnée de m véhicules identiques, chacun de capacité Q est disponible au dépôt. Le but est de déterminer un ensemble de m tournées qui minimise le coût total de transport et qui respecte les contraintes suivantes :
(1) chaque client reçoit une unique visite ;
(2) chaque tournée commence et finit au dépôt ;
(3) la demande total des clients livrés par un même véhicule n'excède pas la capacité du véhicule ;
(4) la longueur de chaque tournée est inférieure à une limite donnée. »
Cette définition généralise le célèbre problème du voyageur de commerce (*Traveling Salesman Problem* - TSP). Le TSP peut être vu comme le cas mono-véhicule incluant uniquement les contraintes (1) et (2) du VRP.

Les définitions suivantes sont plus unanimes.

Capacitated VRP (CVRP) Dans le CVRP les véhicules sont tous identiques, basés à un unique dépôt et disponibles en quantité limitée. On impose uniquement des contraintes sur la capacité des véhicules. L'objectif est de minimiser la longueur des tournées en satisfaisant toutes les demandes.

Distance-Constrained VRP (DVRP) Il s'agit du CVRP dans lequel la contrainte sur la capacité des véhicules est remplacée par une contrainte sur la longueur maximale des tournées.

Heterogeneous fleet VRP (HVRP) Le HVRP est une variante du CVRP dans laquelle différents types de véhicules sont disponibles. Chaque type de véhicule a une

capacité, un coût fixe d'utilisation et un coût variable, dépendant de la distance parcourue, qui lui sont propres. Le nombre de véhicules de chaque type est limité.

Vehicle Fleet Mix Problem (VFMP) Le VFMP est un HVRP dans lequel le nombre de véhicules de chaque type est illimité.

Split Delivery VRP (SDVRP) Le SDVRP est un CVRP dans lequel les clients peuvent être livrés en plusieurs fois. La flotte de véhicules peut être limitée ou illimitée.

VRP with Time Window (VRPTW) Le VRPTW est un CVRP dans lequel on impose des dates de passage chez les clients. Chaque client i est alors associé à une fenêtre de temps $[a_i, b_i]$ et, éventuellement, à un temps de service. Le véhicule doit arriver pour effectuer la livraison à une date comprise dans la fenêtre de temps.

VRP with Backhauls (VRPB) Le VRPB est une extension du CVRP dans laquelle l'ensemble des clients est séparé en deux sous-ensembles A et B. Le premier contient les clients à livrer, tandis que le second contient les clients à collecter. Pour chaque tournée visitant à la fois des clients de A et de B, les clients de A doivent être visités en premier (le véhicule part plein du dépôt pour livrer A et revient plein au dépôt de marchandises collectées chez B).

VRP with Pickups and Deliveries (VRPPD) Le VRPPD est un problème particulier car il porte le nom de VRP mais relève en réalité plus du *pickup and delivery* (voir chapitre 3). Dans ce problème, les marchandises doivent être collectées chez un client (appelé en général origine) pour en livrer un autre (appelé en général destination). Une origine est associée à une unique destination. Notons que, dans les problèmes classiques de tournées de véhicules (*vehicle routing problems*), les clients doivent être, soit livrés avec des marchandises initialement entreposées au dépôt, soit collectés pour ramener leurs marchandises au dépôt. Cependant, dans le VRPPD, les échanges se font entre clients.

Site-Dependant VRP (SDVRP) Dans le *Site-Dependant* VRP, on dispose d'une flotte hétérogène de véhicules disponibles en quantité limitée. Chaque client ne peut être visité que par certains types de véhicules. Contrairement au HVRP, on ne considère pas de coûts fixes sur l'utilisation des véhicules et les coûts de transports sont indépendants du type de véhicule.

Multi-Compartment Vehicle Routing Problem (MC-VRP) Dans le MC-VRP on dispose d'une flotte homogène de véhicules (limitée ou illimitée) pour livrer m types de marchandises. Chaque véhicule possède m compartiments de capacité limitée et le compartiment i est réservé à la marchandise i. Chaque client doit se faire livrer une ou plusieurs marchandises dans une quantité donnée. Chaque type de marchandises doit être livrée avec le même véhicule mais des marchandises différentes peuvent être livrées avec des véhicules différents.

Le tableau 4.1 synthétise les contraintes communément étudiées pour les différentes variantes du VRP et situe le problème du voyageur de commerce (*Traveling salesman Problem* - TSP) par rapport à celles-ci.

Dénomination du Problème	visite unique	Capa. limitée	long. limitée	flotte homogène	flotte hétérogène	flotte limitée	véh. unique	fenêtres de temps	autre	Remarques	référence
TSP Travelling Salesman Problem	×					×	×			-	
VRP Vehicle Routing Problem	×	×	(×)	×		(×)				Les définitions du VRP varient d'un ouvrage à l'autre : Prins [Pri09b] ignore la contrainte de longueur max des tournées et considère que la flotte peut être illimitée ; [TV02] évoque la possibilité de plusieurs contraintes opérationnelles sans en préciser le type	[CLSV07, TV02, Pri09b]
CVRP Capacitated VRP	×	×		×		×				-	[CLSV07, TV02]
DVRP Distance Constrained VRP	×		×	×		×				-	[TV02]
HVRP Heterogeneous fleet VRP	×	×			×	×				possibilité coûts fixes et coût fonction de la distance parcourue par les véhicules	[Pri09b]
VFMP Vehicle Fleet Mix Problem	×	×			×					même remarque que pour HVRP	[Pri09b]
SDVRP Split Delivery VRP		×		×		(×)				-	[PCF08]
VRPTW VRP with Time Window	×	×		×		×		×		-	[TV02]
VRPB VRP with Backhauls	×	×		×		×			×	2 catégories de clients : fournisseurs et collecteurs, on doit passer voir tous les fournisseurs après les collecteurs	[TV02]
VRPPD VRP with pickups and deliveries	×	×		×		×			×	2 catégories de clients : fournisseurs et collecteurs, un fournisseur est associé à un unique collecteur	[TV02]
SDVRP (2) Site-Dependant VRP	×	×			×	×			×	il existe des contraintes de dépendance entre les clients et le type de véhicule	[BBV08]
MCVRP Multi Compartment VRP		×		×		(×)			×	les véhicules ont des compartiments de capacité limitée correspondant chacun à un type de marchandises	[PCF08]

TAB. 4.1 – Synthèse des contraintes rencontrées dans les différents problèmes de tournées de véhicules sur nœuds

Les différentes colonnes du tableau 4.1 ont la signification suivante :
- « visite unique »: les clients sont visités une seule fois, par un seul véhicule ;
- « capa. limitée »: la capacité des véhicules est limitée ;
- « long. limitée »: il existe une longueur maximale à ne pas dépasser pour les tournées ;
- « flotte homogène »: tous les véhicules sont identiques ;
- « flotte hétérogène »: il existe plusieurs types de véhicules ;
- « flotte limitée »: les véhicules sont disponibles en quantité limitée ;
- « véh. unique »: la flotte est réduite à un unique véhicule ;
- « fenêtres de temps »: il y a des restrictions sur les dates de passage chez les clients ;
- « autre »: le problème possède d'autres contraintes (voir la colonne remarque) ;
- « remarques »: remarques sur le problème et ses éventuelles contraintes additionnelles ;
- « référence »: article ou ouvrage sur lequel nous nous sommes appuyé pour la définition donnée.

4.2 Nouvelle approche pour la résolution du *two-dimensional Loading Capacitated Vehicle Routing Problem*

Parmi les problèmes de tournées, certains se situent à la frontière des problèmes d'ordonnancement car ils incluent ou comprennent des sous-problèmes de type ordonnancement. Parmi ceux-ci, le 2L-CVRP occupe une place à part car :
- il s'agit d'un problème récent introduit par Iori *et al.* ([ISGV07]) en 2007 ;
- il regroupe deux problèmes reconnus comme difficiles : le *Capacitated Vehicle Routing Problem* (CVRP) et le *Two Orthogonal Packing Problem* (2OPP).

Plus précisément, le 2L-CVRP est un problème de tournées de véhicules dans lequel des colis définis par leur poids et leurs dimensions doivent être livrés à des clients. Pour cela, on dispose d'une flotte de véhicules tous identiques et de capacité limitée (ils possèdent une limite sur le poids total des colis qu'ils peuvent transporter). Ces véhicules sont de dimension fixée, ainsi le chargement des colis est soumis à deux types de contraintes :
- contrainte sur le poids total des colis transportés ;
- contrainte sur le placement en deux dimensions des colis dans le véhicule (les colis doivent être placés dans le véhicule sans se superposer).

L'objectif est de déterminer des tournées de sorte que toutes les demandes soient satisfaites et que la distance parcourue par les véhicules soit minimale.

Le 2L-CVRP combine donc deux problèmes de logistique : le problème de *Capacitated Vehicle Routing* (CVRP) et le problème d'*orthogonal packing* en deux dimensions (2OPP).

4.2.1 Définition du problème et état de l'art

4.2.1.1 Le *Capacitated Vehicle Routing Problem*

Le CVRP est un problème de tournées de véhicules qui a été largement étudié ces dernières années ([BBV08], [TV02], [Pri09b]). Il est défini par un graphe complet $G = (X, E)$ où

- $X = \{0, 1, \ldots, n\}$ est un ensemble de sommets dans lequel 0 représente le dépôt et l'ensemble $\{1, \ldots, n\}$ représente l'ensemble des clients ;
- E est l'ensemble des arcs. Chaque arc (i, j) est associé à un coût correspondant à la distance entre i et j.

Notons que chaque client étant associé à une demande, il y a n demandes. Une flotte de K véhicules identiques de capacité Q est localisée au dépôt. L'objectif est de déterminer un ensemble de tournées tel que la distance totale parcourue par les véhicules soit minimale et toutes les demandes soient satisfaites. Une tournée doit commencer et finir au dépôt. Un client doit être livré par un seul véhicule : la préemption sur les demandes est interdite.

La résolution des instances de CVRP de taille moyenne ou grande est limitée aux métaheuristiques comme il est spécifié dans [JGH+05].

4.2.1.2 Le *two-dimensional Loading Capacitated Vehicle Routing Problem*

Le 2L-CVRP est une extension du CVRP qui inclut des contraintes de chargement en deux dimensions. Ce problème a été introduit par Iori *et al.* ([ISGV07]) en 2007. Il est particulièrement bien adapté pour représenter les problèmes de logistique liés à la distribution de biens. La figure 4.1 illustre une solution du 2L-CVRP à deux véhicules : le premier véhicule livre les clients C_1, C_2, C_3 et C_4 et le second livre les clients C_5 et C_6.

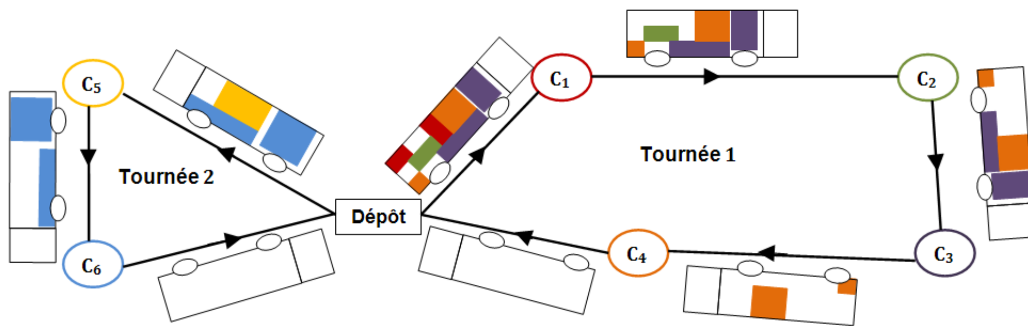


FIG. 4.1 – Exemple de tournées pour le 2L-CVRP

Afin de définir les contraintes de chargement, les notations suivantes sont ajoutées aux notations précédemment définies pour le CVRP :

- la dimension de l'aire de chargement des véhicules : $\mathcal{A} = L \times l$ où L est la longueur du véhicule et l sa largeur ;
- le nombre m_i de colis destinés au client i, $i = 1 \ldots n$;
- le poids p_i de l'ensemble des colis destinés au client i, $i = 1 \ldots n$;
- la longueur L_{ik} et la largeur l_{ik} de chaque colis $k \in [1 \ldots m_i]$ d'un client i, $i = 1 \ldots n$;
- les tournées $t = \{t_0, t_1, \ldots, t_{n(t)}, t_{n(t)+1}\}$ composées des $n(t)$ clients t_1 à $t_{n(t)}$, (t_0 et $t_{n(t)+1}$ représentent le dépôt.

Une tournée t est valide pour le 2L-CVRP si :

- elle vérifie les contraintes de capacité : $\sum_{i=1}^{n(t)} p_{t_i} \leq Q$;
- elle vérifie les contraintes de placement : les colis sont placés dans le véhicule sans se chevaucher.

Une solution du 2L-CVRP est un ensemble de tournées valides qui satisfont toutes les demandes.

Le 2L-CVRP peut être soumis à différents types de contraintes de chargement :

chargement séquentiel : les colis doivent être placés dans le véhicule suivant l'ordre de passage chez les clients. Ainsi, un colis du client j ne doit pas être placé devant un colis du client i si le client i est servi avant le client j. Cette contrainte facilite le déchargement ;

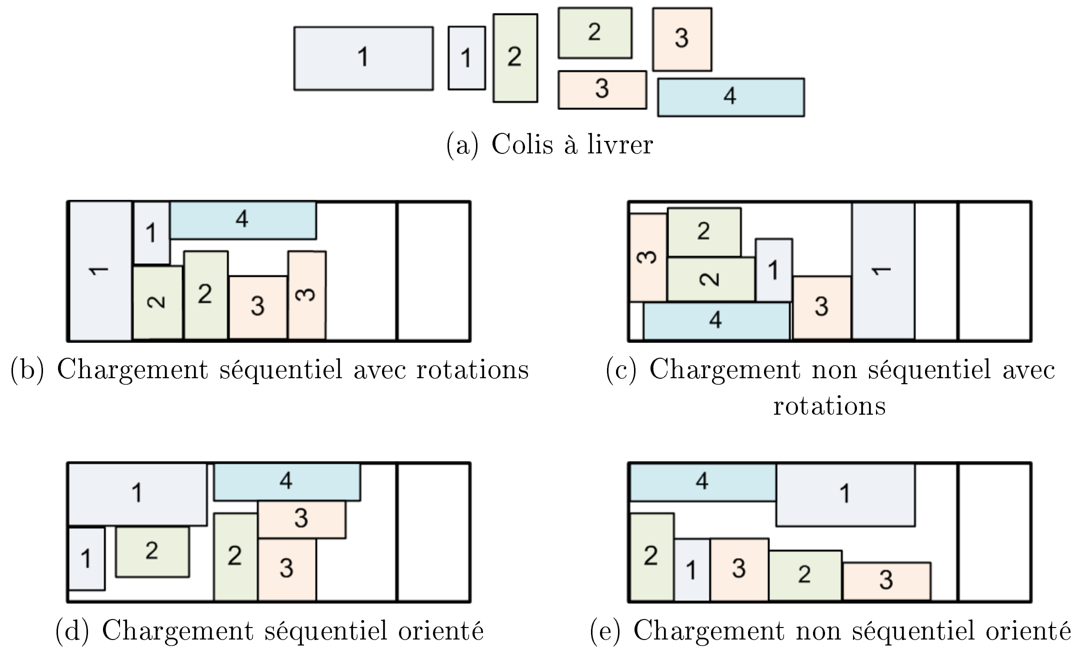
chargement non séquentiel : les colis sont placés dans le véhicule dans un ordre quelconque ;

chargement avec rotations : les colis peuvent subir une rotation de 90 ° avant d'être placés dans le véhicule. Cette contrainte permet de placer plus de colis dans un même véhicule ;

chargement orienté : les colis ne peuvent pas subir de rotation pour être placés dans un véhicule.

Il y a donc quatre possibilités pour charger les véhicules :
- chargement séquentiel et avec rotations ;
- chargement non séquentiel et avec rotations ;
- chargement séquentiel et orienté ;
- chargement non séquentiel et orienté.

La figure 4.2 illustre ces quatre possibilités pour une tournée de quatre clients servis dans l'ordre 1, 2, 3 et 4.



(a) Colis à livrer

(b) Chargement séquentiel avec rotations

(c) Chargement non séquentiel avec rotations

(d) Chargement séquentiel orienté

(e) Chargement non séquentiel orienté

Fig. 4.2 – Les quatre possibilités de chargement des véhicules (le numéro sur les colis correspond au numéro du client auquel ils doivent être livrés)

Dans [FDHI07] Fuellerer *et al.* proposent d'étendre la notation 2L-CVRP pour prendre en compte les contraintes considérées. On distingue alors quatre problèmes :

le 2|SR|L-CVRP : 2L-CVRP avec chargement séquentiel et rotations (*Two dimensional Sequential Rotated Loading*) ;

le 2|UR|L-CVRP : 2L-CVRP avec chargement non séquentiel et avec rotations (*Two dimensional Unrestricted Rotated Loading*) ;

le 2|SO|L-CVRP : 2L-CVRP avec chargement séquentiel et orienté (*Two dimensional Sequential Oriented Loading*) ;

le 2|UO|L-CVRP : 2L-CVRP avec chargement non séquentiel et orienté (*Two dimensional Unrestricted Oriented Loading*).

Notons que dans une publication plus récente ([FDHI09]), les auteurs ont modifié ces notations (le S de *Sequential* est devenu R pour *Rear* et le R de *Rotated* est devenu N pour *Non-oriented*). Nous préférons garder les notations initiales qui semblent plus intuitives et s'adaptent mieux à la terminologie française.

Iori *et al.* ([ISGV07]) sont les premiers à traiter le 2L-CVRP. Ils proposent une résolution exacte grâce à un *branch and cut*. Leur approche est limitée aux instances de petite taille (moins de 25 clients). Ils considèrent seulement le chargement séquentiel et orienté. Gendreau *et al.* ([GILM08b]) proposent un algorithme avec recherche tabou. Leur algorithme permet de résoudre des instances de plus grande taille et il est adapté aux chargements séquentiels et non séquentiels dans le cas orienté. Zachariadis *et al.* ([ZTK09]) proposent pour ce problème une métaheuristique composée d'une recherche tabou et d'une recherche locale guidée. Ils traitent également le chargement séquentiel et non séquentiel dans le cas orienté. Enfin, récemment Fuellerer *et al.* ([FDHI09]) ont proposé un algorithme de colonie de fourmis pour résoudre le 2L-CVRP. Ils sont les premiers à proposer la possibilité de faire subir une rotation aux colis pour le chargement. Leur algorithme permet donc de prendre en compte n'importe laquelle des quatre configurations de chargement. A notre connaissance, leur méthode est celle qui fournit les meilleurs résultats.

Très récemment, Leung *et al.* ([LZZZ10]) ont proposé un algorithme de recherche tabou étendu. Ils s'intéressent uniquement au cas orienté (2|SO|L-CVRP et 2|UO|L-CVRP). Ils utilisent des techniques proches de celles développées par Zachariadis *et al.* et les complètent par de nouvelles méthodes, notamment pour le problème de chargement (placement en 2D). Ils montrent qu'ils obtiennent ainsi de meilleurs résultats que Zachariadis *et al.*.

Le 3L-CVRP (CVRP avec contrainte de chargement en trois dimensions) a également été étudié par [GILM06].

Dans ce chapitre, nous proposons une nouvelle approche pour résoudre le 2|UR|L-CVRP et le 2|UO|L-CVRP, c'est-à-dire le CVRP avec contraintes de chargement non séquentiel dans les cas avec et sans rotation. Cette approche repose sur un schéma de résolution générale de type GRASP×ELS. La stratégie de résolution est la même pour le 2|UR|L-CVRP et le 2|UO|L-CVRP. Les algorithmes impliqués dans la construction des tournées et la recherche locale sont communs aux deux problèmes. La seule différence réside dans la vérification des contraintes de chargement (voir section 4.2.2.8).

4.2.2 Le GRASP×ELS pour le 2|UR|L-CVRP et le 2|UO|L-CVRP

4.2.2.1 Principe général de la méthode proposée

Le 2L-CVRP est un problème qui combine le CVRP et le 2OPP. Comme on l'a vu au chapitre 2, section 2.5, le 2OPP peut être relaxé en un problème de RCPSP. A partir de cette constatation, on propose de résoudre le 2L-CVRP en deux phases :

Phase 1 : la contrainte de chargement liée au placement des colis est relaxée par une contrainte de type RCPSP (le sous-problème de 2OPP est remplacé par sa relaxation en RCPSP, voir chapitre 2, section 2.5). Les tournées doivent alors vérifier les contraintes de capacité et les contraintes de RCPSP pour être valides. Ce nouveau problème, qui est une relaxation du 2L-CVRP, est nommé RCPSP-CVRP ;

Phase 2 : on considère les meilleures solutions trouvées pour le RCPSP-CVRP (celles qui réalisent la plus petite distance parcourue par les véhicules). On tente de transformer pour chaque tournée la solution du RCPSP en une solution du 2OPP. Si, pour toutes les tournées, la solution du RCPSP a été transformée en une solution du 2OPP avec succès, alors la solution définie par l'ensemble des tournées est une solution du 2L-CVRP.

Le RCPSP-CVRP est résolu par une métaheuristique GRASP×ELS. Durant la phase de recherche locale, de nombreuses solutions du RCPSP-CVRP sont explorées. Pour chacune, on teste la validité des tournées. Le test de validité est donc exécuté un grand nombre de fois, c'est pourquoi il doit être rapide. L'intérêt de la relaxation du 2OPP en RCPSP vient du fait que le RCPSP est moins contraint. Il est donc plus facile de trouver une solution *via* des heuristiques. De plus, comme on l'a vu dans le chapitre 2, s'il existe une solution du RCPSP, alors, il existe presque toujours une solution du 2OPP. Par ailleurs, la transformation de la solution du RCPSP en solution du 2OPP est très rapide.

Durant la phase de résolution du RCPSP-CVRP, les N_{best} meilleures solutions sont sauvegardées. A la fin de la résolution, on tente de transformer la meilleure de ces solutions en solution du 2L-CVRP. Si on échoue, on recommence avec la solution suivante et ainsi de suite jusqu'à réussir une transformation. Le RCPSP étant très souvent transformable en 2OPP, on est assuré, en choisissant N_{best} suffisamment grand, de toujours trouver une solution du 2L-CVRP au terme de ce processus.

Pendant la phase d'optimisation, on peut être amené à construire des solutions avec un nombre de tournées plus grand que le nombre de véhicules disponibles. On ajoute alors une pénalité P au coût d'une telle solution. Le coût $f(S)$ d'une solution S est donc calculé de la manière suivante :

$$f(S) = \sum_{t \in t(S)} f(t) + P.(K(S) - K)^{+}$$

où $f(t)$ est le coût de la tournée t, $K(S)$ est le nombre de véhicules utilisés dans la solution S et $t(S)$ est l'ensemble des tournées associées à la solution S.

A la fin du processus, seules les solutions respectant le nombre de véhicules disponibles sont considérées. La figure 4.3 récapitule les différentes étapes de résolution.

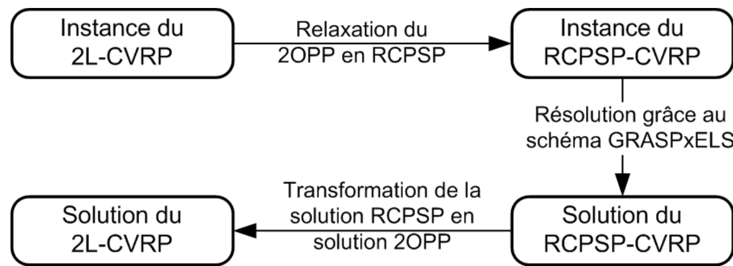


FIG. 4.3 – Principe général de résolution pour le 2L-CVRP

4.2.2.2 Principe du GRASP×ELS

Comme on l'a vu dans le chapitre 1, la métaheuristique GRASP×ELS est une hybridation du GRASP et de l'ELS. Ce schéma est particulièrement intéressant car il permet d'échapper, dans une certaine mesure, aux bassins d'attraction constitués par les minima locaux. En effet, le GRASP permet d'explorer différentes régions de l'espace des solutions grâce à des démarrages multiples. L'ELS permet d'intensifier la recherche dans la région courante grâce à des mutations et à une recherche locale sur la solution courante. Son principe de fonctionnement général est rappelé figure 4.4. L'abréviation « RL » désigne la recherche locale.

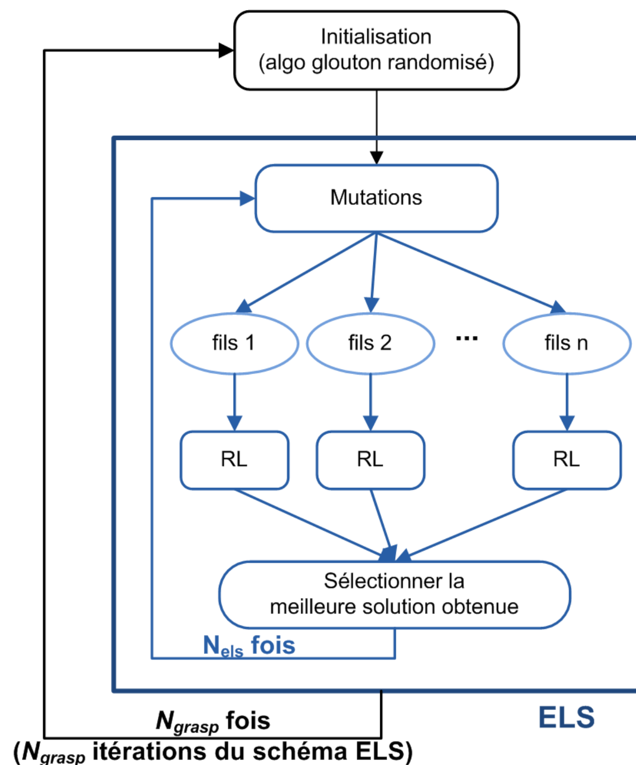


FIG. 4.4 – Schéma GRASP×ELS

4.2.2.3 Stratégie de recherche dans l'espace des solutions

On utilise deux espaces de recherche différents. Le premier est celui des solutions (ensemble des tournées) et le second est celui des « tours géants » définis comme la concaténation des tournées d'une solution. La procédure *Split* permet de convertir un tour géant en un ensemble de tournées solution du RCPSP-CVRP. Le *Split* a été utilisé dans de nombreux problèmes de tournées de véhicules comme, par exemple, le *Capacitated Arc Routing Problem* ([LPRC04]), le *Vehicle Routing Problem* ([Pri04]) ou encore le *Location Routing Problem* ([DLPP10]). La procédure *Split* est détaillée section 4.2.2.6. Le GRASP×ELS tire profit des deux espaces de recherche en recherchant à la fois dans l'espace des tours géants et dans l'espace des solutions du RCPSP-CVRP. Notons que dans [Pri04], Prins obtient de très bons résultats en alternant entre ces deux espaces de recherche. Sa proposition s'inscrit dans la même logique que le schéma que nous proposons.

Ainsi, durant le GRASP×ELS, un tour géant T est transformé en solution du RCPSP-CVRP grâce à la procédure *Split* qui assure que les tournées générées respectent les contraintes de capacité et de RCPSP. Une procédure *Concat* permet de concaténer les tournées d'une solution en tour géant. Le tour géant ainsi généré peut être de nouveau transformé grâce au *Split*. Ce processus permet d'alterner entre les deux espaces de recherche. Il est illustré par la figure 4.5.

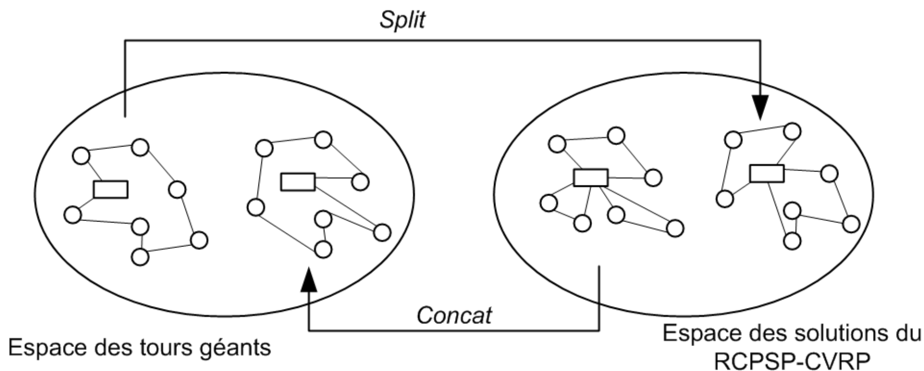


Fig. 4.5 – Alternance entre deux espaces de recherche

L'algorithme 46 présente le schéma de résolution générale du RCPSP-CVRP. Dans ce schéma, l'algorithme `GénérerSolution` de génération de solutions initiales utilise des versions randomisées de l'heuristique de Clarke et Wright [CW64] et de l'heuristique *Path Scanning* [GDB83] (voir section 4.2.2.4). Pour les instances de grande taille, le GRASP×ELS inclut une procédure d'intensification : si la solution initiale S_i générée par l'heuristique `GénérerSolution` est plus mauvaise que la meilleure solution S^*, alors on remplace la solution S_i par S^*. Ceci a pour effet d'intensifier la recherche dans la région qui semble prometteuse. Il est clair que cette phase d'intensification arrive d'autant plus souvent que S^* est de bonne qualité. Ce processus se révèle très efficace pour les instances de grande taille. Sur les instances de petite ou moyenne taille, il est préférable de privilégier la diversification en gardant la solution fournie par `GénérerSolution`.

Algorithme 46 : GRASP×ELS pour la résolution du RCPSP-CVRP

Entrées :
N_{grasp} //nombre d'itérations du GRASP
N_{els} //nombre d'itérations de l'ELS
N_{mut} //nombre de mutations
N_{max} //nombre maximal d'itérations sans amélioration
Sorties : les N_{best} meilleures solutions trouvées

```
1  f(S*) ← +∞ ;
2  //GRASP :
3  Pour i = 1 à N_grasp faire
4  |   S ←GénérerSolution() ;
5  |   T ←Concat(S) ;
6  |   Si f(S) < f(S*) alors
7  |   |   S* ← S ;
8  |   Si intensification = vrai alors
9  |   |   S* ← S* ;
10 |   j ← 0; k ← 0 ;
11 |   //ELS :
12 |   Tant que j < N_els et k < N_max faire
13 |   |   j ← j + 1 ;
14 |   |   f'' ← +∞ ;
15 |   |   Pour m = 1 à N_mut faire
16 |   |   |   T' ←Mutation(T) ;
17 |   |   |   S' ← Split(T') ;
18 |   |   |   S' ← RechercheLocale(S') ;
19 |   |   |   T' ← Concat(S') ;
20 |   |   |   Si f(S') < f'' alors
21 |   |   |   |   f'' ← f(S') ;
22 |   |   |   |   T'' ← T' ;
23 |   |   |   |   S'' ← S' ;
24 |   |   Si f(S'') ≥ f(S) //on n'a pas amélioré alors
25 |   |   |   k ← k + 1 ;
26 |   |   Si f(S'') < f(S*) //nouvelle meilleure solution alors
27 |   |   |   S* ← S''
28 |   |   T ← T'' // le meilleur fils devient solution courante
```

4.2.2.4 Génération d'une solution initiale

Le point de départ du GRASP×ELS est la génération d'une solution initiale. Afin d'assurer une bonne performance du GRASP, l'heuristique utilisée doit être capable de fournir des solutions très diversifiées. Nous utilisons donc, pour générer les solutions initiales, quatre méthodes différentes. Ainsi, la procédure `GénérerSolution` utilise itérativement quatre heuristiques : heuristique de Clarke et Wright randomisée (CW), *Path Scanning* (PS), *Path Scanning* randomisée (PSR) et heuristique de génération aléatoire (GR). No-

tons que l'heuristique de Clarke et Wright et *Path Scanning* sont des heuristiques souvent utilisées dans les problèmes de tournées de véhicules car elles sont assez efficaces et faciles à randomiser.

Les heuristiques ne prennent pas en compte les contraintes de RCPSP mais calculent l'aire totale occupée par les colis et limitent cet aire à $\alpha \times \mathcal{A}$ où $\mathcal{A}$ est la surface de chargement disponible et α est un coefficient inférieur ou égal à 1. Les tournées de la solution générée sont transformées en tour géant, puis de nouveau divisées en utilisant la procédure *Split*. La procédure *Split* crée des tournées valides pour le RCPSP-CVRP (le nombre de tournées peut éventuellement être supérieur au nombre de véhicules).

4.2.2.4.1 Heuristique de Clarke et Wright (CW)

L'heuristique de Clarke et Wright consiste, tout d'abord, à créer autant de tournées que de clients. Le nombre de tournées est ensuite réduit itérativement en fusionnant les tournées. Un algorithme similaire, nommé *Augment-Merge*, destiné aux problèmes d'*Arc Routing* a été proposé par Golden *et al.* ([GDB83]).

L'algorithme de Clarke et Wright se déroule en deux étapes :

Etape 1 : une tournée par client est créée ;

Etape 2 : les tournées sont considérées par paire. Pour chaque paire, on calcule le gain engendré par la fusion des deux tournées. Le gain est la différence entre la distance parcourue avant fusion et la distance parcourue après fusion. Pour deux tournées $t = \{t_1, \ldots, t_n\}$ et $s = \{s_1, \ldots, s_m\}$ quatre fusions sont envisageables :
- la tournée t puis la tournée s : $\{t_1, \ldots, t_n, s_1, \ldots, s_m\}$;
- la tournée s puis la tournée t : $\{s_1, \ldots, s_m, t_1, \ldots, t_n\}$;
- la tournée t à l'envers puis la tournée s : $\{t_n, \ldots, t_1, s_1, \ldots, s_m\}$;
- la tournée t puis la tournée s à l'envers : $\{t_1, \ldots, t_n, s_m, \ldots, s_1\}$.

La fusion qui a le plus grand gain est réalisée. On recommence avec les nouvelles tournées jusqu'à obtenir le nombre de tournées désirées ou que plus aucune fusion ne puisse être réalisée (à cause de la contrainte de capacité des véhicules par exemple).

L'algorithme de Clarke et Wright est parfois appelé « méthode de la marguerite » à cause de la solution initiale qui a une allure de marguerite (voir figure 4.6).

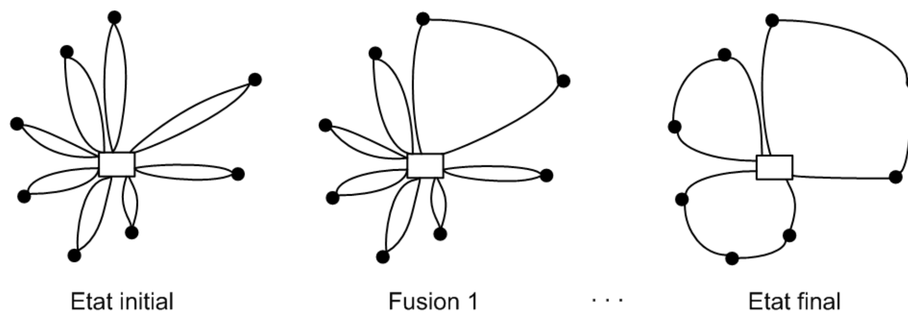


Fig. 4.6 – Etat des tournées à différentes étapes de l'algorithme de Clarke et Wright

Nous utilisons une version randomisée de l'algorithme de Clarke et Wright dans laquelle la fusion à réaliser est choisie aléatoirement parmi les meilleures fusions possibles.

4.2.2.4.2 *Path Scanning* (PS)

Path Scanning a été développée par Golden *et al.* ([GDB83]) pour les problèmes de tournées sur arcs. Dans cette heuristique, les tournées sont construites une à une. Lors de la construction d'une tournée, les clients sont ajoutés un à un. Le choix du client à ajouter dans la tournée en construction est soumis à cinq critères :

(1) maximiser la distance du client au dépôt ;
(2) minimiser la distance du client au dépôt ;
(3) minimiser la productivité (quotient entre la quantité à livrer et la distance à parcourir) ;
(4) maximiser la productivité (quotient entre la quantité à livrer et la distance à parcourir) ;
(5) si le véhicule est chargé à moins de la moitié de sa capacité, appliquer le critère (1) et, sinon, appliquer le critère (2).

Dans la proposition de Golden *et al.*, l'algorithme est exécuté cinq fois : une fois par critère.

Nous proposons une version modifiée de cet algorithme, mieux adaptée aux contraintes de chargement de notre problème : entre autre nous prenons en compte la dimension des colis. Nous considérons sept critères pour le choix du client à insérer :

(1) client qui maximise la distance du client au dépôt ;
(2) client qui minimise la distance du client au dépôt ;
(3) client dont le poids des colis à livrer est le plus faible ;
(4) client dont le poids des colis à livrer est le plus grand ;
(5) client dont l'aire total des colis à livrer est le plus faible ;
(6) client dont l'aire total des colis à livrer est le plus grand ;
(7) appliquer le critère (1) si l'aire de chargement du véhicule est à moitié pleine, appliquer le critère (2) sinon.

Les deux premiers critères sont identiques à ceux de Golden *et al.*, les autres ont été adaptés. Les critères (3) et (4) prennent en compte le poids des colis. Les critères (5) et (6) prennent en compte les dimensions des colis. Le critère (7) incite le véhicule à retourner vers le dépôt lorsqu'il est à moitié plein. L'algorithme est exécuté avec un critère donné. Une version randomisée (PSR) permet de choisir le client aléatoirement parmi les meilleurs clients selon le critère considéré.

4.2.2.4.3 Heuristique de génération aléatoire (GR)

L'algorithme de génération aléatoire construit un tour géant de manière complètement aléatoire : le tour est initialement vide, à une itération k un client est choisi aléatoirement et inséré dans le tour en position k. Notons que, contrairement aux heuristiques précédentes, on construit un tour géant et non un ensemble de tournées.

4.2.2.4.4 Description de la méthode `GénérerSolution`

La méthode de génération d'une solution initiale exploite les quatre algorithmes : Clarke et Wright randomisé (CW), *Path-Scanning* (PS), *Path-Scanning* randomisé (PSR), génération aléatoire (GR).

Algorithme 47 : GénérerSolution

Entrées : courant, critère, N_{gen}
Sorties : courant, critère, S

1 $cpt \leftarrow 0$;
2 $i \leftarrow 0$;
3 **Tant que** $i \leq N_{gen}$ *et* $t(S) > K$ *//nombre de tournées supérieur au nombre de véhicules* **faire**
4   **cas où** *courant = 1*
5         $S \leftarrow$ **CW**() ;
6         $cpt \leftarrow cpt + 1$;
7   **Si** $cpt > 2$ **alors**
8             $courant \leftarrow courant + 1$;
9             $cpt \leftarrow 0$;
10   **cas où** *courant = 2*
11        $S \leftarrow$ **PS**(critère) ;
12   critère←critère + 1 ;
13   **Si** *critère > 7* **alors**
14   courant ← courant + 1 ;
15   critère ← 1 ;
16   **cas où** *courant = 3*
17        $S \leftarrow$ **PSR**(critère) ;
18        $cpt \leftarrow cpt + 1$;
19   **Si** $cpt > 2$ **alors**
20   critère←critère + 1 ;
21            $cpt \leftarrow 0$;
22   **Si** *critère > 7* **alors**
23   courant ← courant + 1 ;
24   critère ← 1 ;
25   **cas où** *courant = 4*
26        $S \leftarrow$ **GR**() ;
27        $cpt \leftarrow cpt + 1$;
28   **Si** $cpt > 2$ **alors**
29            $courant \leftarrow courant + 1$;
30            $cpt \leftarrow 0$;
31    $i \leftarrow i + 1$;
32    $T \leftarrow$ **Concat**(S) ;
33    $S \leftarrow$ **Split**(T) ;

Les heuristiques CW, PS et PSR ne prennent pas en compte les contraintes de RCPSP mais calculent l'aire totale occupée par les colis et limitent cet aire à $\alpha \times \mathcal{A}$. Après construction, les tournées sont transformées en tour géant. Cette étape est bien sûr inutile pour l'heuristique de génération aléatoire qui construit directement un tour géant.

A la fin de la génération du tour géant, la procédure *Split* est exécutée et crée des tournées valides pour le RCPSP-CVRP. Le nombre de tournées créées par *Split* peut être supérieur au nombre de véhicules. Une pénalité est alors ajoutée (voir section 4.2.2.1). Ces heuristiques sont exécutées plusieurs fois, jusqu'à trouver une solution qui respecte le nombre de tournées autorisées ou qu'un nombre maximal d'itérations ait été atteint. La procédure `GénérerSolution` est donnée par l'algorithme 47.

Grâce à l'utilisation de quatre heuristiques différentes, la procédure `GénérerSolution` génère des solutions d'une grande diversité ce qui est profitable au GRASP.

4.2.2.5 Mutation

L'opérateur de mutation agit sur un tour géant $T = \{T_1, \dots, T_{n(T)}\}$ qui résulte de la concaténation des tournées T_1 à $T_{n(T)}$. L'opérateur de mutation agit en deux temps :

1. un point de coupure est choisi aléatoirement entre deux tournées T_i et T_{i+1}. Les deux ensembles de tournées $\{T_1, \dots, T_i\}$ et $\{T_{i+1}, \dots, T_{n(T)}\}$ sont échangés de sorte que T devient $\{T_{i+1} \dots T_{n(T)}, \dots, T_1, \dots, T_i\}$;
2. deux clients sont choisis aléatoirement dans T et leur position est échangée.

Exemple 9. *Par exemple, on considère les tournées* $T_1 = \{1, 2, 3\}$, $T_2 = \{4, 5\}$ *et* $T_3 = \{6, 7, 8, 9\}$. *On a alors le tour géant* $T = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. *L'opérateur de mutation peut transformer* T *de la manière suivante :*

1. *le point de coupure est choisi entre la tournée* T_1 *et la tournée* T_2. T *devient* $\{4, 5, 6, 7, 8, 9, 1, 2, 3\}$ *;*
2. *les clients 6 et 2 sont choisis.* T *devient* $\{4, 5, 2, 7, 8, 9, 1, 6, 3\}$.

Finalement $T = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ *a été transformée en* $T = \{4, 5, 2, 7, 8, 9, 1, 6, 3\}$.

4.2.2.6 Split

La transformation des tours géants en un ensemble de tournées solutions du RCPSP-CVRP est réalisée par la procédure `Split_RCPSP` dérivée de la procédure *Split*. Nous décrivons dans un premier temps la procédure *Split* classique, puis nous proposons une adaptation qui prend en compte les contraintes de RCPSP.

4.2.2.6.1 Le *Split* classique

La procédure *Split* a été proposée par Beasley en 1983 ([Bea83]). Il l'utilise dans le cadre de la stratégie de résolution « route-first, cluster-second » pour les problèmes de tournées de véhicules (*VRP*). La stratégie « route-first, cluster-second » a été reprise et utilisée par de nombreux auteurs. De manière générale, cette stratégie est constituée de deux phases :

Phase 1 : Transformation du problème de tournées en un problème de voyageur de commerce. Cette transformation consiste à relaxer les contraintes qui obligent à avoir plusieurs tournées : par exemple les contraintes de capacité des véhicules, les contraintes de longueur maximale sur les tournées . . . La résolution de ce problème relaxé aboutit à un « tour géant »: tournée qui commence au dépôt, visite tous les clients puis finit au dépôt ;

Phase 2 : Découpage du tour géant en plusieurs tournées qui respectent les contraintes *via* la construction d'un graphe auxiliaire. Ce découpage respecte l'ordre des clients. Il est réalisé de manière exacte (pour l'ordre des clients considéré) par la procédure *Split*.

Soit $\lambda = \{\lambda_1, \ldots, \lambda_n\}$ la séquence des clients décrite par le tour géant construit durant la phase 1. La procédure *Split* consiste alors à construire un graphe auxiliaire de $n + 1$ sommets numérotés de 0 à n où 0 représente le dépôt et les sommets 1 à n représentent les clients. Il existe un arc du sommet $i = \lambda_{i'}$ vers le sommet $j = \lambda_{j'}$ si et seulement si i est avant j dans le tour géant. Dans ce cas, l'arc (i, j) représente la séquence $\{\lambda_{i'}, \ldots, \lambda_{j'}\}$ du tour géant. Il est valué par le coût de la tournée $\{\lambda_{i'}, \ldots, \lambda_{j'}\}$, c'est-à-dire $d(0, \lambda_{i'}) + d(\lambda_{i'}, \lambda_{i'} + 1) + \ldots + d(\lambda_{j'} - 1, \lambda_{j'}) + d(\lambda_{j'}, 0)$.

La figure 4.7 montre le graphe auxiliaire associé au tour géant $ABCDE$. Pour cet exemple, on considère que les véhicules sont disponibles en nombre illimité et que les contraintes de capacité sont toujours respectées (les distances sont choisies pour avoir au moins un découpage). Les valeurs sur les arcs du graphe auxiliaire représentent la tournée considérée et son coût.

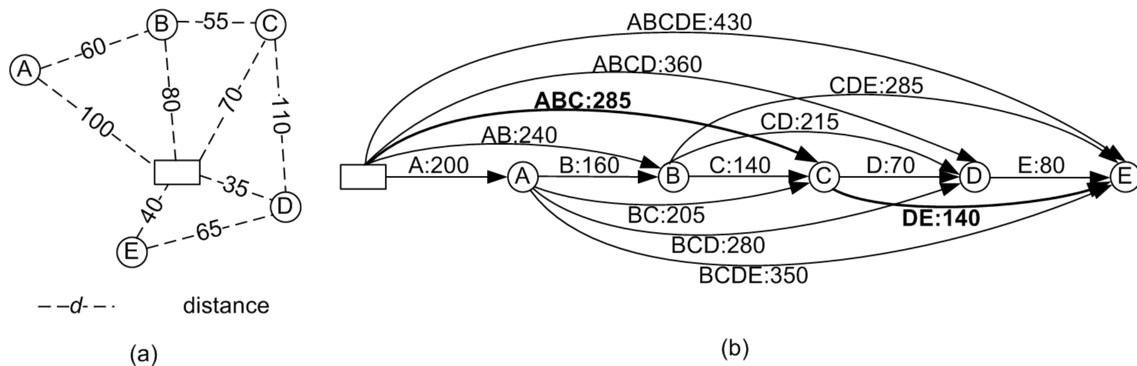


FIG. 4.7 – (a) Distances entre les sommets ; (b) Graphe auxiliaire associé au SPLIT

Les tournées optimales (pour la séquence de clients considérée) sont obtenues par un calcul du plus court chemin pour aller de 0 à n. Sur l'exemple de la figure 4.7 le plus court chemin entre 0 et $n = E$ est donné par les arcs $(0, C)$ et (C, E) ce qui correspond aux tournées ABC et DE. Le tour géant $ABCDE$ est donc découpé de manière optimale en deux tournées : ABC et DE.

En pratique, on évite le calcul de tous les arcs ainsi que le calcul du plus court chemin en apposant des labels sur les sommets du graphe auxiliaire. Un label sur un sommet $j = \lambda_{j'}$ représente une solution partielle incluant les clients de rang λ_1 à $\lambda_{j'}$.

Dans le cas du *Split* classique (un seul type de contrainte), un label comprend :

– le coût : distance totale parcourue par les tournées de la solution partielle ;
– le label père : indique le dernier label de la tournée précédente.

Comme plusieurs découpages sont possibles pour arriver à un sommet j, le sommet j peut avoir plusieurs labels. On applique alors des règles de dominance pour garder le meilleur label (ou un ensemble des meilleurs labels). Dans le cas où le nombre de véhicules disponibles est illimité, un label domine un autre simplement s'il est de meilleur coût. Il est clair que plus il y a de contraintes à prendre en compte dans les labels (capacité, nombre limité de véhicules...), plus la règle de dominance est complexe (voir section suivante).

La solution est donnée par le meilleur label (celui de coût le plus faible) apposé sur le sommet n. Le découpage optimal est calculé en considérant la succession des labels pères.

La figure 4.8 (a) montre les labels calculés pour le graphe auxiliaire de la figure 4.7. Dans un souci de simplification, les contraintes de capacité sont supposées toujours satisfaites. Sur chaque sommet seul le label de meilleur coût est conservé. Les labels dominés sont indiqués mais rayés par une croix. Dans cet exemple, un label contient un entier qui correspond au coût et une lettre qui correspond au sommet sur lequel est apposé le label père. Par exemple, le label $(355, C)$ sur le sommet D signifie que le coût de la solution partielle est 355 et que le label père est celui de C. Le label père nous indique que la tournée courante commence juste après C : elle commence donc en D. Comme elle finit aussi en D il s'agit de la tournée qui contient uniquement D. Pour connaître la tournée précédente, il faut regarder le label père du label $(285, 0)$ apposé sur le sommet C : il s'agit de celui donné par 0 (dépôt). On en déduit que la tournée précédente commence juste après 0 donc en A : il s'agit alors de la tournée ABC.

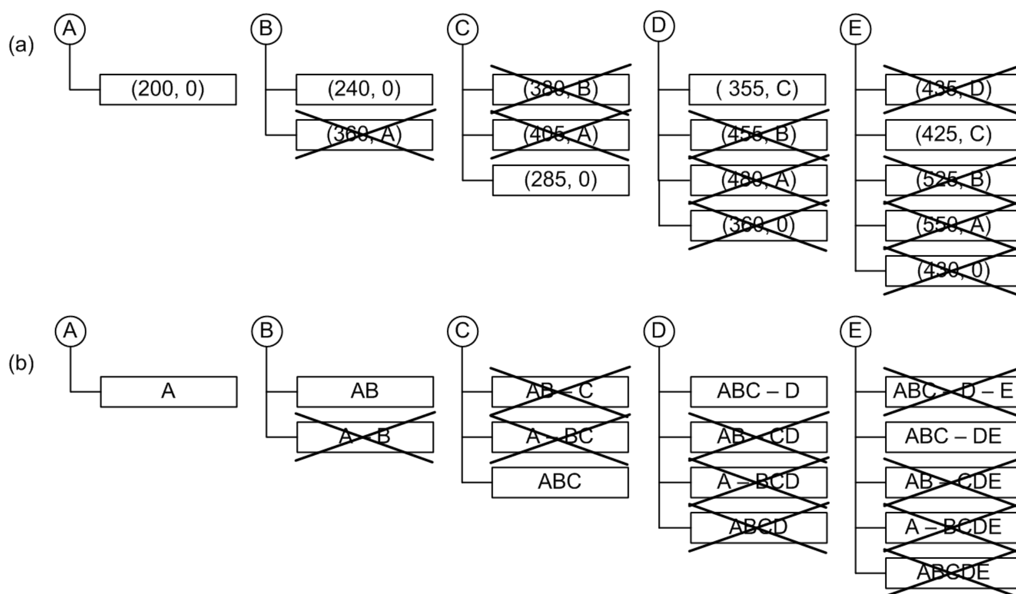


FIG. 4.8 – (a) Labels associés au SPLIT ; (b) Tournées associées aux labels

La figure 4.8 (b) reproduit les labels précédents en les remplaçant par les tournées auxquelles ils correspondent : deux tournées sont séparées par le signe $-$. Ainsi, l'information $ABC - D$ associée au label $(355, C)$, signifie que le label $(355, C)$ représente la solution partielle donnée par les tournées ABC et D.

La figure 4.9 illustre le tour géant et les tournées après découpage par la procédure *Split*.

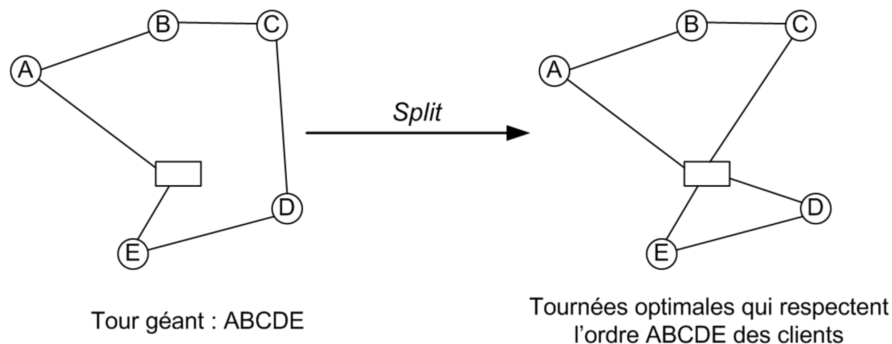


Fig. 4.9 – Découpage du tour géant avec *Split*

La procédure *Split* a déjà été appliquée à de nombreux problèmes de tournées de véhicules. On peut citer [Pri04] dans lequel le *Split* est appliqué à des problèmes de *VRP*. La méthode utilisée est la meilleure publiée à ce jour pour les instances de Christophides et de Golden. [LPRC04] utilise le Split dans le cadre d'un algorithme mémétique pour résoudre le *CARP*. La méthode fait partie des meilleures méthodes publiées. La meilleure méthode pour la résolution du HVRP ([Pri09b]) contient également le *Split*. D'autres problèmes, comme le *MC-VRP* ([PCF08]), *LRP*([DLPP10]), *HFVRP* ([SS97]) ou encore le *Truck and Trailer Routing Problem* ([VMP+09]) ont également été résolus avec succès grâce au *Split*.

4.2.2.6.2 Le *Split* modifié pour le RCPSP-CVRP

Dans le cadre du RCPSP-CVRP, le *Split* a été adapté pour prendre en compte les contraintes de RCPSP. La démarche générale est la même que celle décrite précédemment. Dans le graphe auxiliaire les arcs sont développés seulement s'ils respectent les contraintes de capacité et de RCPSP. La figure 4.10 montre le graphe auxiliaire associé au tour géant $ABCDE$ en considérant 3 véhicules de capacité 10. Le poids des colis à livrer pour chaque client est indiqué entre parenthèses. On suppose, sur cet exemple, que les contraintes de RCPSP sont respectées. En pratique, ces contraintes sont vérifiées pour chaque tournée.

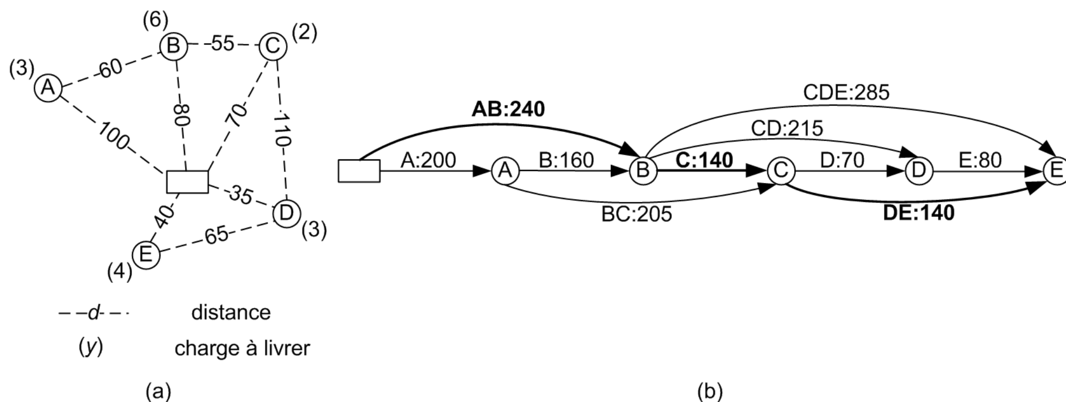


Fig. 4.10 – (a) Distances entre les sommets et charges à livrer par 3 véhicules de capacité 10 ; (b) Graphe auxiliaire associé au *Split* avec contrainte de capacité

Les labels et la règle de dominance doivent êtres adaptés. Le $l^{ème}$ label L_j^l associé au sommet j est constitué des information suivantes :

- nombre de véhicules restant (K_j^l) ;
- coût de la solution partielle courante (C_j^l) ;
- label père.

Il n'est plus possible, comme précédemment, de garder un unique label par sommet à cause des contraintes sur le nombre limité de véhicules. De ce fait, le label père est défini par deux valeurs : le sommet sur lequel il est apposé et son rang parmi les labels de ce sommet. Le $l^{ème}$ label du sommet j nommé L_j^l est donc défini par $L_j^l = \left(K_j^l,\ C_j^l,\ k,\ i\right)$ où i est le sommet sur lequel est apposé le label père et k est son rang parmi les labels de i.

Un label L_j^l domine un label $L_j^{l'}$ si :

$$\left(K_j^l > K_j^{l'}\ et\ C_j^l \leq C_j^{l'}\right)\ \ ou\ \ \left(K_j^l \geq K_j^{l'}\ et\ C_j^l < C_j^{l'}\right)$$

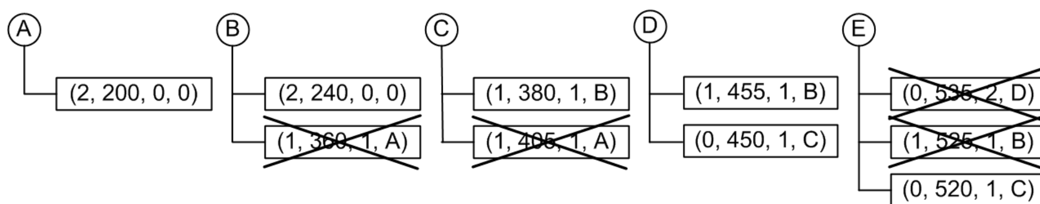


Fig. 4.11 – Labels associés au graphe auxiliaire de la figure 4.10

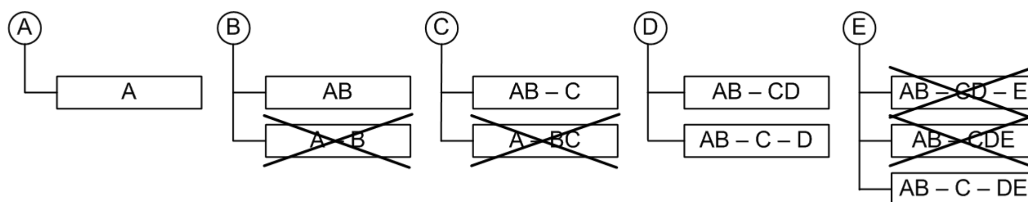


Fig. 4.12 – Tournées associées aux labels de la figure 4.11

Sur le sommet final, le meilleur label est celui de plus faible coût, la règle de dominance ne s'applique pas sur ce sommet. La figure 4.11 illustre le calcul des labels pour le graphe auxiliaire de la figure 4.10 et la figure 4.12 donne les solutions partielles associées aux labels.

La procédure `Split-RCPSP` que nous utilisons est décrite par l'algorithme 48. La procédure `test_RCPSP` utilisée par cet algorithme renvoie vrai si la tournée est RCPSP-réalisable, faux sinon. La procédure `test_RCPSP` est détaillée section 4.2.2.8. Notons que si on omet la vérification du RCPSP dans la procédure `Split-RCPSP`, on obtient la procédure de *Split* classique pour des problèmes de CVRP.

Algorithme 48 : Split-RCPSP

Entrées : T //tour géant
Sorties : S //solution pour le RCPSP-CVRP

```
1  L0 ← (K, 0, 0, 0) ;
2  Pour i = 0 à n − 1 faire
3      j ← i + 1 ; t ← ∅ ; client ← 0 ;
4      Répéter
5          Pc ← c //client précédent le client courant ;
6          c ← c + 1 //client courant;
7          t ← t ∪ c //tournée courante;
8          //Mettre à jour la charge Qt et le coût Ct de la tournée courante :
9          Si j = i+1 //c est l'unique élément dans t alors
10             Qt ← pc ; Ct ← d(0, c) + d(c, 0) ;
11         sinon
12             Qt ← Qt + pc ; Ct ← Ct + d(Pc, c) + d(c, 0) − d(Pc, 0) ;
13         stop ← (Qt > Q) ou (test_RCPSP(t) = faux) ;
14         Si stop = faux alors
15             pour chaque label L_i^k = (K_i^k, C_i^k, k, i) de i faire
16                 Soit p le nombre de labels déjà apposés sur j ;
17                 L = (K_i^k − 1, C_i^k + Ct, p + 1, j) //label de j dérivant de L_i^k ;
18                 Si L non dominé par un autre label de j alors
19                     Insérer L dans l'ensemble des labels de j ;
20         j ← j + 1 ;
21     jusqu'à stop = vrai ou j > n ;
22 S ← solution déduite du meilleur label de n ;
```

4.2.2.7 Recherche Locale

Le but de la recherche locale est d'améliorer une solution du RCPSP-CVRP. La recherche locale repose sur deux opérateurs classiques : le *2-opt* et le *swap*. Ces opérateurs existent en deux versions : dans une même tournée ou entre deux tournées.

La figure 4.13 illustre l'opérateur d'échange dans une même tournée : les sommets A et B de la tournée $BACD$ sont échangés. La figure 4.14 illustre l'opérateur d'échange entre deux tournées : le sommet A de la tournée AFG est échangé avec le sommet E de la tournée $EBCD$.

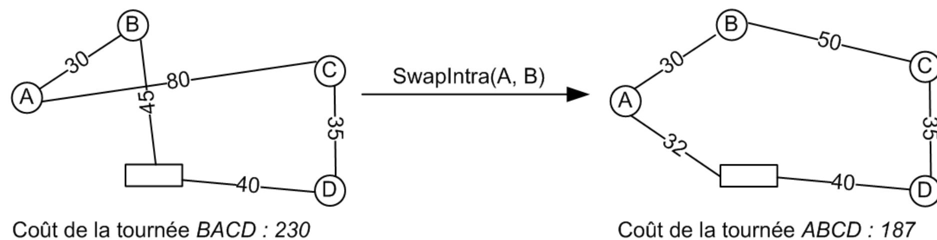


FIG. 4.13 – Illustration de l'opérateur SwapIntra

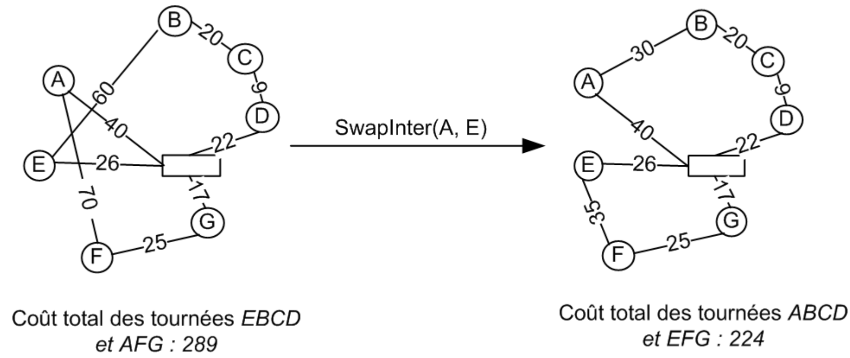


FIG. 4.14 – Illustration de l'opérateur `SwapInter`

La figure 4.15 illustre l'opérateur 2-opt dans une même tournée : les arcs (A, D) et (B, E) sont supprimés et remplacés par les arcs (A, B) et (D, E). La figure 4.16 illustre l'opérateur 2-opt entre deux tournées : l'arc (B, F) de la tournée $ABFE$ et l'arc (C, G) de la tournée $DCGF$ sont supprimés. Ils sont remplacés par les arcs (B, C) et (F, G) afin de former les nouvelles tournées $ABCD$ et $EFGH$.

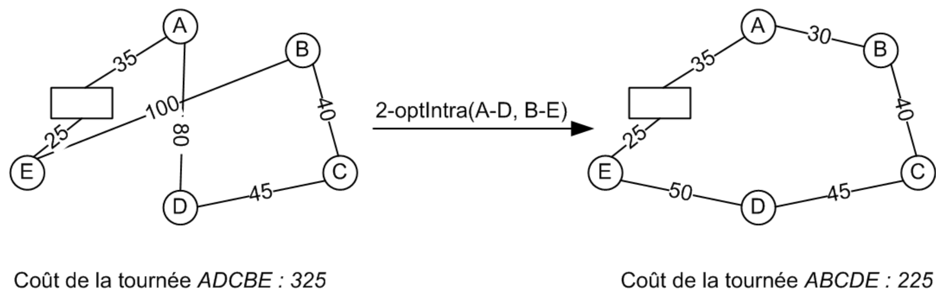


FIG. 4.15 – Illustration de l'opérateur `2-OptIntra`

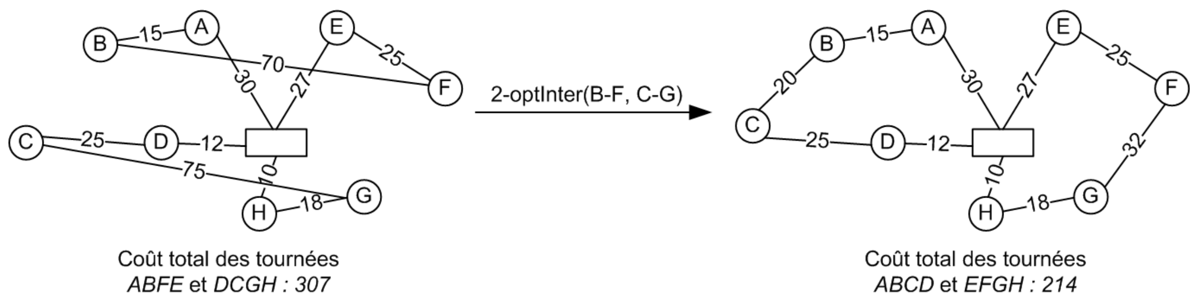


FIG. 4.16 – Illustration de l'opérateur `2-OptInter`

Ces quatre opérateurs sont utilisés dans un schéma de descente. On utilise donc quatre procédures de descente `RL-SwapIntra`, `RL-SwapInter`, `RL-2-OptIntra` et `RL-2-OptInter` au sein de la recherche locale. L'ordre d'appel de ces procédures est randomisé de sorte qu'une itération de la recherche locale termine, soit par une manipulation entre deux tournées, soit par une manipulation dans une même tournée. L'algorithme de principe est donné par algorithme 49. Une constante $SEUIL$ fixe l'amélioration minimale à obtenir pour continuer la recherche locale. En pratique, elle est égale à 1.

Algorithme 49 : Recherche Locale

```
    Entrées : S, N_RL, N_split, SEUIL
    Sorties : S
 1  cpt ← 0 ;
 2  cpt2 ← 0 ;
 3  Répéter
 4  |  f_sav ← f(S) ;
 5  |  S ← RL-2-OptIntra() ;
 6  |  S ← RL-2-OptInter() ;
 7  |  Si random < 0.5 //random : nombre aléatoire entre 0 et 1 alors
 8  |  |  S ← RL-SwapIntra() ;
 9  |  |  S ← RL-SwapInter() ;
10  |  sinon
11  |  |  S ← RL-SwapInter() ;
12  |  |_ S ← RL-SwapIntra() ;
13  |  Si f_sav − f(S) ≤ SEUIL et cpt2 < N_split alors
14  |  |  S ← RL-Split() ;
15  |  |_ cpt2 ← cpt2 + 1 ;
16  |  cpt ← cpt + 1 ;
17  jusqu'à cpt ≥ N_RL ou f_sav − f(S) ≤ SEUIL ;
```

Si les quatre procédures de descente n'améliorent pas suffisamment la solution, une cinquième procédure `RL-Split` est exécutée. Cette procédure consiste à concaténer, dans un ordre aléatoire, les tournées pour obtenir un tour géant. Ce tour géant est ensuite redécoupé en tournées grâce au *Split*. Le but est de trouver un meilleur découpage des tournées. Comme cette procédure fait appel au *Split*, elle est assez coûteuse en temps de calcul. C'est pourquoi elle n'est utilisée que si la solution n'a pas été suffisamment améliorée et un nombre de fois limité. En pratique le nombre d'appels à cette procédure est limité à $N_{split} = 2$.

Nous avons choisi de nous limiter aux opérateurs de transformation locale *2-Opt* et *Swap* pour deux raisons :

- ces opérateurs fournissent des schémas de descente peu coûteux en temps de calcul ;
- ils donnent de très bons résultats utilisés dans le cadre du GRASP×ELS.

4.2.2.8 Vérification du RCPSP pour les chargements avec et sans rotation

Le problème de RCPSP vient du problème d'*orthogonal packing* qui a été relaxé. On utilise, pour résoudre ce problème, l'algorithme `SGS_RCPSP(`A`, `l_C`)` décrit chapitre 2, section 2.5, avec les entrées suivantes :

- A est l'ensemble des activités qui dérivent de l'ensemble des colis à placer dans le véhicule : la consommation d'une activité correspond à la largeur du colis associé et la durée correspond à sa longueur. La quantité totale de ressources disponibles est égale à la largeur du véhicule ;
- L_C (makespan à ne pas dépasser) est la longueur du véhicule.

Cet algorithme étant randomisé, on l'exécute plusieurs fois pour essayer de trouver une solution au RCPSP. Notons que cet algorithme est défini pour le cas orienté (2|UO|L) mais s'adapte facilement au cas avec rotations (2|UR|L), comme il a été montré dans la section 2.5. C'est donc cet algorithme qui est utilisé pour les deux cas traités.

4.2.2.9 Stratégie pour accélérer la vérification du RCPSP

Durant le GRASP×ELS de très nombreuses tournées sont évaluées. Parmi celles-ci, beaucoup ne sont pas RCPSP-réalisables, c'est pourquoi il est nécessaire d'adopter une stratégie qui permet de détecter rapidement les tournées non RCPSP-réalisables afin de ne pas exécuter inutilement l'algorithme `SGS_RCPSP` un grand nombre de fois. Les stratégies décrites ci-après sont appliquées aussi bien dans le cas avec ou sans rotation, sauf l'utilisation de la borne inférieure qui est réservée au cas sans rotation.

4.2.2.9.1 Utilisation de l'aire de chargement

Soit une tournée t. Un premier test très simple et très rapide consiste à vérifier que l'aire totale occupée par les colis ne dépasse pas l'aire de chargement $\mathcal{A}$:

$$\sum_{i \in t} \sum_{k=1}^{m_i} L_{ik}.l_{ik} \leq \mathcal{A}$$

De plus, on suppose que si l'aire totale occupée par les colis est trop proche de $\mathcal{A}$ il n'existe pas de solution au RCPSP. On réduit alors l'aire de chargement grâce à un coefficient $\alpha \leq 1$ et on impose :

$$\sum_{i \in t} \sum_{k=1}^{m_i} L_{ik}.l_{ik} \leq \alpha.\mathcal{A}$$

4.2.2.9.2 Utilisation d'une borne inférieure

Avant d'exécuter l'algorithme `SGS_RCPSP`, on calcule la borne inférieure LB3 de Klein et Scholl [KS99b] et on vérifie qu'elle est bien inférieure ou égale à la longueur du véhicule. Cette borne inférieure considère les activités *incompatibles*, c'est-à-dire les activités qui ne peuvent pas être planifiées en parallèle à cause des contraintes de ressources. Cette borne est donc utilisée uniquement dans le cas du chargement orienté (2|UO|L). En effet, dans le cas du chargement avec rotation, on ne peut pas prévoir a priori le sens des colis et donc la consommation des activités.

4.2.2.9.3 Relaxation de la longueur du véhicule

Si les deux conditions précédentes sont satisfaites, la procédure `SGS_RCPSP` est exécutée plusieurs fois jusqu'à ce qu'une exécution donne une solution au RCPSP de makespan inférieur ou égal à la longueur du véhicule $L = L_C$ ou qu'un nombre maximal d'itérations soit atteint. Afin d'éviter un nombre d'itérations de la méthode trop important, on fixe une valeur maximale du makespan à ne pas dépasser en fonction du nombre d'exécutions déjà

réalisées. Ainsi le meilleur makespan calculé durant les 10 premières exécutions ne doit pas dépasser 125% de la longueur du véhicule. Si les 10 premières exécutions n'ont pas fourni de solution de makespan inférieur ou égal à L mais qu'une solution de makespan inférieur ou égal à $1.25 \times L$ a été trouvée, on continue les exécutions de `SGS_RCPSP`. Dans le cas contraire, on arrête en supposant qu'il n'existe pas de solution. Il y a ainsi quatre étapes à franchir comme décrit par l'algorithme 50.

Algorithme 50 : test_RCPSP

Entrées :
A //ensemble des activités associées aux colis de la tournée t
l_C //longueur du véhicule
N_1, N_2, N_3, N_4 //nombre d'itérations maximales par étape
Sorties : res //booléen indiquant si la tournée est RCPSP-réalisable

1 $res \leftarrow faux$;
2 $N \leftarrow N_1$;
3 $cpt \leftarrow 0$;
4 **Tant que** $res = faux$ *et* $cpt < N$ **faire**
5 $(res, C_{max}) \leftarrow$ **SGS_RCPSP**(A, l_C) ;
6 **cas où** $cpt = N_1$ *et* $C_{max} \leq 1.25 \times l_C$
7 $N \leftarrow N_2$;
8 **cas où** $cpt = N_2$ *et* $C_{max} \leq 1.15 \times l_C$
9 $N \leftarrow N_3$;
10 **cas où** $cpt = N_3$ *et* $C_{max} \leq 1.10 \times l_C$
11 $N \leftarrow N_4$;

4.2.2.9.4 Fonction de hachage

Dans le cas du chargement non séquentiel qui nous intéresse, plusieurs tournées différentes peuvent être similaires du point de vue de la réalisabilité du RCPSP. C'est le cas des tournées qui visitent les mêmes clients. Par exemple, les tournées $T_1 = \{1, 2, 3, 4\}$ et $T_2 = \{3, 2, 4, 1\}$ sont différentes car elles ne visitent pas les clients dans le même ordre, mais, si une des deux tournées est RCPSP-réalisable, alors l'autre l'est également. En effet, les véhicules associés à ces tournées visitent les mêmes clients donc ils ont les mêmes colis à livrer. Les deux véhicules peuvent donc être chargés de la même manière.

Afin d'éviter plusieurs tests de RCPSP-réalisabilité pour des tournées similaires (qui visitent les mêmes clients), un conteneur associatif sauvegarde les tournées déjà testées avec le résultat du test de RCPSP-réalisabilité. Un conteneur associatif est une structure de données qui associe une clé à une donnée. Dans notre cas, la donnée est une tournée t et la clé associée à t est un entier calculé grâce à une fonction de hachage $h(t)$. Nous utilisons le conteneur associatif *map* de la *Standard Template Library* (STL) pour C++ car il possède une structure d'arbre rouge noir (*red black tree*), ce qui permet une recherche et une insertion en temps logarithmique d'un élément.

La fonction de hachage $h(t)$ est définie par :

$$h(t) = \prod_{i \in t} prem(i) \mod N_{map}$$

où $prem(i)$ désigne le $i^{ème}$ nombre premier, mod désigne l'opérateur modulo et N_{map} la taille de la *map*.

Définition : Deux tournées t et t' qui contiennent les mêmes clients sont dites *RCPSP-équivalentes*.

Propriété : Deux tournées t et t' RCPSP-équivalentes ont la même valeur de hachage, c'est-à-dire $h(t) = h(t')$.

Cette propriété est intéressante car elle implique qu'un seul test de réalisabilité pour le RCPSP peut être effectué pour deux tournées RCPSP-équivalentes et un seul résultat est stocké.

Cependant, il est clair que la présence du modulo dans le calcul de la valeur de hachage implique que deux tournées non RCPSP-équivalentes peuvent avoir la même valeur. Dans ce cas on dit qu'il y a collision. Afin d'avoir une procédure la plus rapide possible, on sauvegarde en priorité dans la *map* les tournées dont la vérification de la réalisabilité du RCPSP a été suffisamment coûteuse. Pour cela, on définit un nombre minimal N_{min} d'itérations de la procédure `test_RCPSP` en dessous duquel la tournée n'est pas sauvegardée dans la *map*. De cette façon, la sauvegarde des tournées dont le test de validité est le plus coûteux en temps de calcul est privilégiée. Cette stratégie permet de réduire considérablement le temps de calcul passé au test de RCPSP-réalisabilité des tournées.

Au cours du GRASP×ELS, on sauvegarde donc les résultats obtenus par la procédure `test_RCPSP` dans un tableau *val*. Ce tableau prend les valeurs :

$$val[h(t)] = \begin{cases} 0 & \text{si} \quad \text{la RCPSP-réalisabilité de } t \text{ n'a pas encore été testée} \\ 1 & \text{si} \quad t \text{ est RCPSP-réalisable} \\ -1 & \text{si} \quad t \text{ n'est pas RCPSP-réalisable} \end{cases}$$

Finalement, la procédure utilisée pour vérifier la RCPSP-réalisabilité des tournées est décrite par l'algorithme 51.

4.2.2.10 Transformation de la solution du RCPSP-CVRP en une solution du 2L-CVRP

Au terme du GRASP×ELS, les N_{best} meilleures solutions du RCPSP-CVRP ont été sauvegardées. On essaie alors de transformer les solutions du RCPSP-CVRP en solutions du 2L-CVRP, de la meilleure à la moins bonne, jusqu'à trouver une solution transformable. Pour transformer ces solutions, il suffit de calculer une solution au placement pour chaque tournée. Pour cela, on utilise la procédure `ResolutionOPP` avec $itMax1 = 50\ 000$ et $itMax2 = 2\ 000 \times |A|$ (où $|A|$ est le nombre de colis à placer).

Algorithme 51 : test_RCPSP_rapide

Entrées :
t //tournée
A //ensemble des activités associées aux colis de la tournée t
L //longueur du véhicule
Sorties : res //booléen indiquant si la tournée est RCPSP-réalisable

```
1  Si val[h(t)] = 0 alors
2  |  (res, nb) ← test_RCPSP(A, L) ;
3  |  Si nb > N_min alors
4  |  |  map[h(t)] ← t ;
5  |  |  Si res = vrai alors
6  |  |  |  val[h(t)] ← 1 ;
7  |  |  sinon
8  |  |  |  val[h(t)] ← −1 ;
9  |  sinon
10 |  |  Si t = map[h(t)] //tournée RCPSP-équivalente alors
11 |  |  |  Si val[h(t)] = 1 alors res ← vrai ;
12 |  |  |  sinon res ← faux ;
13 |  |  sinon
14 |  |  |  res ← test_RCPSP(A, L) ;
```

4.2.3 Expérimentations numériques

Nous comparons nos résultats à ceux des quatre dernières publications :
- Gendreau *et al.*, 2008 ([GILM08b]) qui proposent un algorithme tabou ;
- Zachariadis *et al.*, 2009 ([ZTK09]) qui proposent également un algorithme tabou ;
- Fuellerer *et al.*, 2009 ([FDHI09] et [FDHI07] : rapport technique) qui proposent un algorithme de colonie de fourmis ;
- Leung *et al.*, 2010 ([LZZZ10]) qui proposent un algorithme tabou étendu.

4.2.3.1 Jeux d'instance

Les quatre dernières publications sur le 2L-CVRP utilisent les mêmes jeux de données. Les jeux de données sont identiques pour le 2|UO|L-CVRP et le 2|UR|L-CVRP. Ils se composent de 180 instances réparties dans cinq classes, chaque classe étant composée de 36 instances. Le nombre de colis par client et la dimension des colis d'une instance varient d'une classe à l'autre. La classe 1 correspond à des instances de type CVRP (colis de taille 1×1, c'est-à-dire qu'il n'y a pas de problème de placement à résoudre). Dans les classes i, $i = 2 \ldots 5$, le nombre de colis affectés à chaque client a été choisi suivant une loi uniforme entre 1 et i (voir [GILM08b]). Le nombre de clients d'une instance varie entre 15 et 255, le nombre de véhicules varie entre 3 et 51.

Les instances peuvent être téléchargées à l'adresse suivante : `http://www.or.deis.unibo.it/research.html`. Les solutions trouvées par Fuellerer *et al.* [FDHI09] sont disponibles à l'adresse `http://prolog.univie.ac.at/research/VRPandBPP/`.

4.2.3.2 Conditions expérimentales

A notre connaissance, les meilleurs résultats ont été fournis par Fuellerer *et al.* ([FDHI09]). C'est pourquoi, nous comparons notre méthode à la leur. Nous indiquons également dans les tableaux les résultats de Gendreau *et al.*, Zachariadis *et al.* et Leung *et al.* bien que les conditions expérimentales soient différentes.

Le temps de calcul accordé à la méthode de Fuellerer *et al.* est limité à une heure. La machine utilisée est 1.5 fois plus puissante que celle que nous utilisons. Afin de se placer dans des conditions expérimentales similaires, nous avons multiplié par 1.5 la limite de temps pour le GRASP×ELS. Ainsi, la limite de temps pour le GRASP×ELS est fixée à une heure trente. Le tableau 4.2 donne les conditions expérimentales des quatre dernières publications. Il figure dans ce tableau les informations suivantes :
- caractéristiques du processeur de la machine utilisée (*Proc.*) ;
- le système d'exploitation (*Syst.*) ;
- le langage de programmation (*Lang.*) ;
- la limite de temps de calcul par instance (*Lim.*) ;
- le nombre de runs effectués pour une instance (*Runs*).

	Méthode tabou Gendreau *et al.*	Méthode tabou Zachariadis *et al.*	Colonie de fourmis Fuellerer *et al.*	Méthode tabou Leung *et al.*	GRASP×ELS cette thèse
Proc.	PIV 1.7GHz	PIV 2.4GHz	PIV 3.2 GHz	Core 2 Duo 2.0GHz	Opteron 2.1GHz
Syst.	?	Windows	Linux	Windows	Linux
Lang.	C	C++	C++	C	C++
Lim.	1h	1h	1h	aucune	1h30
Runs	1	200	10	10	10

TAB. 4.2 – Caractéristiques des machines et réglages utilisés pour le 2|UO|L-CVRP et le 2|UR|L-CVRP (ce dernier problème étant traité uniquement par Fuellerer *et al.*)

Toujours dans l'objectif de fournir une comparaison claire des méthodes, nous utilisons le même nombre d'exécutions : 10 exécutions sont réalisées par le GRASP×ELS et la meilleure solution trouvée est reportée dans les tableaux comparatifs. Les résultats détaillés sur chaque instance pour le 2|UO|L-CVRP et le 2|UR|L-CVRP sont disponibles à l'adresse `http://www.isima.fr/~toussain`.

4.2.3.3 Réglage des paramètres

Les paramètres dont la valeur n'a pas été donnée au cours du chapitre dépendent généralement de la taille des instances. Ainsi trois réglages ont été réalisés : un pour les instances de petite taille (instances 1 à 15), un pour les instances de taille moyenne (instance 16 à 19) et un pour les instances de grande taille (instances 20 à 36). Les valeurs de ces paramètres sont reportées dans le tableau 4.3 pour la classe 1 (pas de chargement) et dans le tableau 4.4 pour les classes 2 à 5.

	instances 1 à 15	instances 16 à 19	instances 20 à 36
N_{grasp}	$20 + (n/10) \times 2$	$20 + (n/10) \times 20$	$+\infty$
N_{gen}	5	5	5
P	100 000	100 000	100 000
N_{els}	20	20	20
N_{max}	15	15	15
N_{mut}	10	15	15
N_{RL}	$50 + (n/10)$	$50 + (n/10)$	$50 + (n/10) \times 7$

TAB. 4.3 – Réglage des paramètres pour la classe 1

	instances 1 à 15	instances 16 à 19	instances 20 à 36
N_{grasp}	$20 + (n/10)$	$20 + (n/10)$	$20 + (n/10)$
N_{gen}	5	5	5
P	100 000	100 000	100 000
N_{els}	20	20	20
N_{max}	15	15	15
N_{mut}	10	15	15
N_{RL}	$50 + (n/10)$	$50 + (n/10)$	$50 + (n/10)$
α	0.95 ou 1[1]	0.95 ou 1[1]	0.95 ou 1[1]
N_{map}	1 000 000	1 000 000	1 000 000

TAB. 4.4 – Réglage des paramètres pour les classes 2 à 5

4.2.3.4 Comparaison des résultats

Nous comparons les résultats que nous obtenons avec les quatre derniers articles publiés :

- [GILM08b] (Gendreau *et al.* en 2008) ;
- [ZTK09] (Zachariadis *et al.* en 2009) ;
- [FDHI09] et le rapport technique associé [FDHI07] (Fuellerer *et al.* en 2009) ;
- [LZZZ10] (Leung *et al.* en 2010).

Le 2|UO|L-CVRP (2L-CVRP avec chargement non séquentiel et orienté) est traité par les quatre articles tandis que le 2|UR|L-CVRP (2L-CVRP avec chargement non séquentiel et rotation) est traité uniquement par Fuellerer *et al.*.

4.2.3.4.1 Comparaison des résultats pour le CVRP (classe 1)

Le tableau de résultats 4.5 concernant la classe 1 donne le coût moyen de la solution (distance totale parcourue) pour les 36 instances.

Méthode tabou Gendreau *et al.*	Méthode tabou Zachariadis *et al.*	Colonie de fourmis Fuellerer *et al.*	Méthode tabou Leung *et al.*	GRASP×ELS cette thèse
coût	coût	coût	coût	coût
792.31	777.75	776.04	773.65	**770.77**

TAB. 4.5 – Résultats moyens obtenus sur la classe 1 (CVRP sans chargement)

[0] $\alpha = 0.95$ pour le 2|UO|L-CVRP et $\alpha = 1$ pour le 2|UR|L-CVRP

On constate que le GRASP×ELS donne les meilleurs résultats en moyenne sur la classe 1. Le coût moyen des solutions sur les 36 instances de la classe 1 trouvé par le GRASP×ELS est 770.77 contre 776.04 pour Fuellerer *et al.*, 777.75 pour Zachariadis *et al.*, 792.31 pour Gendreau *et al.* et 773.65 pour Leung *et al.*. De plus, le GRASP×ELS fournit 8 nouvelles meilleures solutions.

A la fin de ce chapitre, le tableau 4.8 donne les résultats détaillés pour les 36 instances (les nouvelles meilleures solutions sont indiquées par un astérisque).

4.2.3.4.2 Comparaison des résultats pour le 2|UO|L-CVRP

Le tableau 4.6 synthétise les résultats obtenus pour les classes (notées *cl.*) 2 à 5 pour le 2|UO|L-CVRP. Les résultats sont comparés, cette fois encore, avec les quatre dernières publications sur le 2L-CVRP car elles traitent toutes le chargement orienté. La ligne « cl. 2-5 » concerne les résultats moyens sur les classes 2 à 5.

	Méthode tabou Gendreau *et al.*	Méthode tabou Zachariadis *et al.*	Colonie de fourmis Fuellerer *et al.*	Méthode tabou Leung *et al.*	GRASP×ELS cette thèse
	coût	coût	coût	coût	coût
cl. 2	-	1205.45	1150.68	1188.38	**1140.44**
cl. 3	-	1217.40	1174.98	1200.14	**1149.14**
cl. 4	-	1223.45	1191.59	1208.52	**1168.25**
cl. 5	-	1078.24	1059.55	1066.17	**1052.29**
cl. 2-5	1216.08	1181.13	1144.20	1165.80	**1127.53**

Tab. 4.6 – Résultats moyens obtenus sur les classes 2 à 5 pour le 2|UO|L-CVRP (chargement non séquentiel et orienté)

Les résultats du tableau 4.6 concernent le 2|UO|L-CVRP, c'est-à-dire le 2L-CVRP avec chargement non séquentiel et orienté (les colis ne peuvent pas subir de rotation). Ils montrent que notre méthode fournit les meilleurs résultats en moyenne sur les classes 2 à 5. De plus, pour chaque classe, la méthode fournit un nombre important de nouvelles meilleures solutions : 21 (sur 36 instances) pour la classe 2, 28 pour la classe 3, 25 pour la classe 4 et 19 pour la classe 5.

A la fin de ce chapitre, le tableau 4.8 donne les résultats détaillés pour les 36 instances pour le 2|UO|L-CVRP (les nouvelles meilleures solutions sont indiquées par un astérisque).

4.2.3.4.3 Comparaison des résultats pour le 2|UR|L-CVRP

Le tableau 4.6 synthétise les résultats obtenus pour les classes 2 à 5 pour le 2|UR|L-CVRP (chargement avec rotations). Les résultats sont comparés uniquement aux résultats de Fuellerer *et al.* car ils sont les seuls à traiter le chargement avec rotations.

Les résultats du tableau 4.6 concernent le 2|UR|L-CVRP, c'est-à-dire le 2L-CVRP avec chargement non séquentiel et rotations autorisées des colis. Seuls Fuellerer *et al.* fournissent des résultats pour ce problème. On constate que notre méthode fournit les meilleurs résultats en moyenne sur les classes 2 à 4. De plus, pour ces classes, la méthode fournit un nombre important de nouvelles meilleures solutions : 19 pour la classe 2, 23 pour la classe 3, 21 pour la classe 4. La classe 5 donne des résultats légèrement moins bons

	Colonie de fourmis Fuellerer *et al.*	GRASP×ELS cette thèse	
	coût	nouv. meilleur	coût
classe 2	1113.44	19/36	**1109.94**
classe 3	1141.25	23/36	**1132.41**
classe 4	1162.75	21/36	**1157.72**
classe 5	**1040.08**	2/36	1052.72
classe 2-5	1114.38		**1113.19**

TAB. 4.7 – Résultats moyens obtenus sur les classes 2 à 5 pour le 2|UR|L-CVRP (chargement non séquentiel et avec rotations)

que ceux de Fuellerer *et al.* en moyenne : le coût moyen des solutions pour cette classe est 1052.72 pour le GRASP×ELS contre 1040.08 pour la méthode de colonie de fourmis de Fuellerer *et al.*. Cependant le GRASP×ELS donne 2 nouvelles meilleures solutions pour la classe 5. En moyenne, sur l'ensemble des classes, le GRASP×ELS fournit les meilleurs résultats.

A la fin de cette section, le tableau 4.9 donne les résultats détaillés pour les 36 instances du 2|UR|L-CVRP (les nouvelles meilleures solutions sont indiquées par un astérisque).

4.2.3.5 Exemples de résultats

Considérons la solution obtenue sur l'instance 1 de la classe 3 dans le cas orienté (les rotations des colis ne sont pas autorisées). La solution obtenue est présentée sur la figure 4.17. Le coût de cette solution est de 284.52.

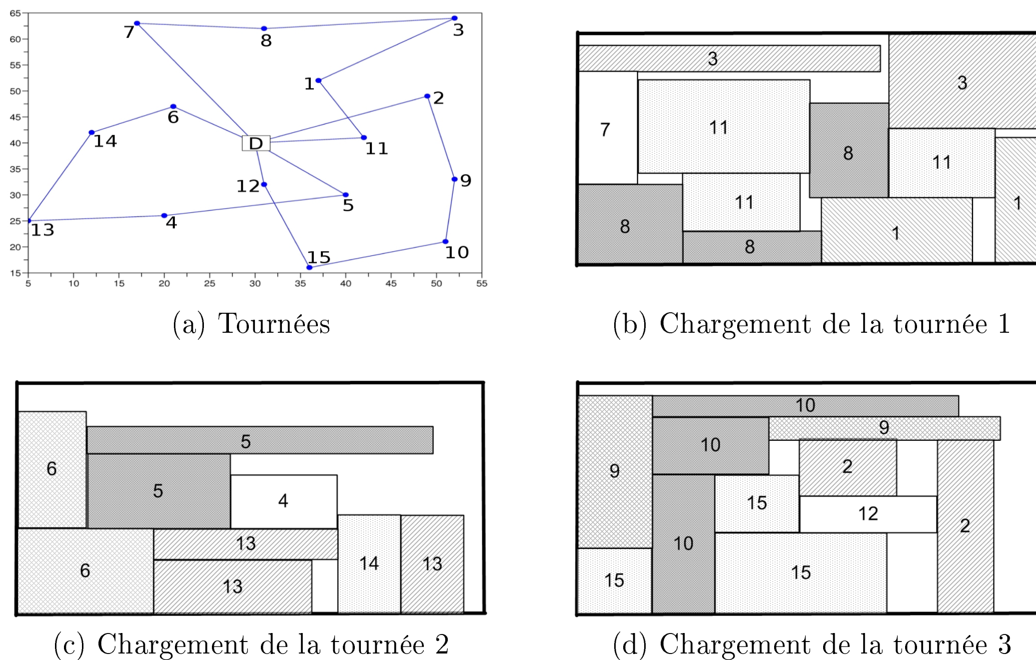
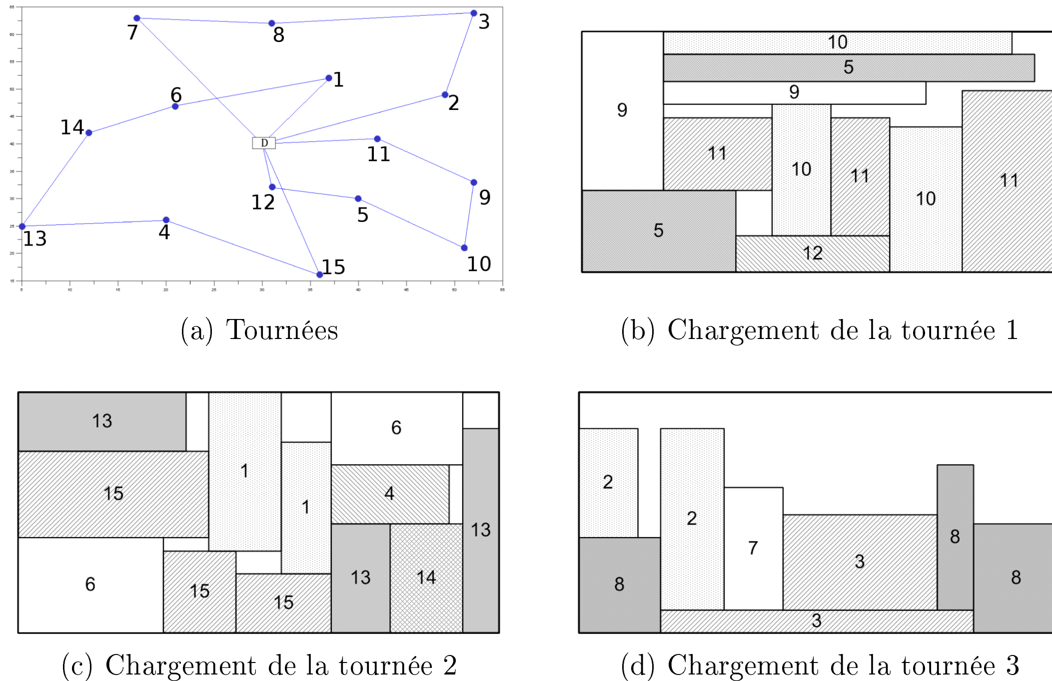


FIG. 4.17 – Résultats sur l'instance 0103 dans le cas orienté (le coût de la solution est 284.52)

En autorisant les rotations, le GRASP×ELS obtient une solution de valeur 284.23. Plusieurs colis ont subi des rotations. En particulier les colis des clients 11 et 6 ont subi

une rotation comme le montre la figure 4.18. Comme on peut le constater, même si les deux solutions sont proches en terme de valeur (284.52 et 284.23) elles sont radicalement différentes en terme de tournées et de chargement des véhicules.



(a) Tournées

(b) Chargement de la tournée 1

(c) Chargement de la tournée 2

(d) Chargement de la tournée 3

Fig. 4.18 – Résultats sur l'instance 0103 dans le cas avec rotations (le coût de la solution est 284.23)

4.3 Conclusion

Ce chapitre aborde les problèmes de tournées de véhicules de type *Vehicle Routing Problem* (*VRP*) et en particulier le 2L-CVRP. Le 2L-CVRP est une extension du CVRP dans laquelle on intègre des considérations géométriques sur la demande des clients : la demande d'un client consiste en un ensemble de colis rectangulaires à placer dans le véhicule, défini par sa surface de chargement. Nous proposons un schéma de type GRASP×ELS pour résoudre ce problème et deux façons d'organiser le chargement : en autorisant la rotation des colis ou non. Le 2L-CVRP intègre donc des sous-problèmes de type placement en deux dimensions. L'originalité de l'approche GRASP×ELS envisagée consiste :

- à relaxer le problème initial en RCPSP-CVRP (relaxation des contraintes de chargement en contraintes de type RCPSP) ;
- à transformer la solution obtenue à la fin du processus d'optimisation en une solution du 2L-CVRP.

L'originalité et la difficulté de résolution du 2L-CVRP repose sur le fait qu'il est nécessaire d'aborder simultanément des problèmes de tournées et d'ordonnancement (placement). Une procédure particulière a été définie pour tester la faisabilité du RCPSP dans des temps de calcul très courts. Cette contrainte de temps de calcul est très forte dans notre schéma d'optimisation car la réalisabilité des tournées est systématiquement testée. L'optimisation repose essentiellement sur deux points :

- l'utilisation d'un GRASP qui constitue à notre connaissance la première application du GRASP à la résolution du 2L-CVRP ;
- l'alternance entre l'espace des solutions du RCPSP-CVRP et l'espace des tours géants tirant avantage de la méthode *Split*.

Le GRASP×ELS a été testé sur des instances de la littérature. Les résultats obtenus sont comparés, pour le cas sans rotation, à ceux de quatre articles récemment publiés. Pour le cas avec rotations, nous nous comparons à un seul des quatre articles, les trois autres ne considérant pas cette possibilité. Les résultats obtenus par notre méthodes sont, en moyenne, les meilleurs publiés. De plus, le GRASP×ELS fournit de nombreuses nouvelles meilleures solutions à la fois pour la classe 1 (CVRP classique) et pour les classes 2 à 5 (CVRP avec contraintes de chargement 2D) pour le cas avec et sans rotation.

Ainsi, le GRASP×ELS montre sa capacité à résoudre le problème de tournées tout en prenant en compte les sous-problèmes de placement 2D qui résultent des contraintes de chargement. Ce type d'approche pourrait s'appliquer à d'autres problèmes de tournées tels que le 3L-CVRP.

	classe 1		classe 2		classe 3		classe 4		classe 5		classe 2-5	
instance	coût	cpu	coût	cpu	coût	cpu	coût	cpu	coût	cpu	coût	cpu
1	278.73	0.00	284.42 *	0.20	284.52 *	0.90	282.95	0.00	278.73	2.50	282.65	0.90
2	334.96	0.00	334.96	0.00	352.16	0.10	334.96 *	0.10	334.96	0.00	339.26	0.10
3	358.40	0.00	387.70	0.80	394.72	0.40	364.45	0.20	358.40	0.40	376.32	0.50
4	430.88	0.00	430.88	0.30	430.88 *	0.30	447.37	0.10	430.88	0.20	435.01	0.20
5	375.28	0.00	375.28	0.10	381.69	0.20	383.87	0.20	375.28	0.10	379.03	0.10
6	495.85	0.00	495.85	0.40	498.16	0.60	498.32	0.50	495.85	0.00	497.04	0.40
7	568.56	0.00	725.46	0.30	678.75 *	0.20	702.45	2.00	657.77 *	3.00	691.11	1.40
8	568.56	0.00	674.55 *	0.20	738.43 *	0.60	692.47	1.60	609.90 *	0.70	678.84	0.80
9	607.65	0.00	607.65	0.20	607.65	0.30	625.10	1.70	607.65	0.20	612.01	0.60
10	535.80	0.00	689.68	6.10	615.68 *	0.80	711.01 *	17.80	686.78 *	35.90	675.79	15.10
11	505.01	0.00	693.45 *	25.90	706.73 *	4.50	786.85 *	10.70	636.77	4.20	705.95	11.30
12	610.00	0.20	610.57	5.40	610.00	54.10	614.23 *	1.60	610.23	6.40	611.26	16.90
13	2006.34	0.00	2585.72 *	105.30	2454.37 *	20.20	2587.63 *	15.50	2334.78 *	171.00	2490.62	78.00
14	837.67	0.20	1038.09 *	177.50	996.25 *	28.10	981.90 *	5.50	921.45 *	108.50	984.42	79.90
15	837.67	0.00	1013.29 *	482.80	1154.66 *	248.90	1234.14 *	55.70	1176.68 *	243.40	1144.69	257.70
16	698.61	0.00	698.61	0.90	698.61	2.90	703.35	12.00	698.61	8.40	699.79	6.00
17	861.79	0.00	870.86	53.10	861.79	2.30	861.79	29.70	861.79	1.30	864.05	21.60
18	723.54	8.30	1004.99 *	885.90	1069.45 *	110.40	1118.71 *	235.00	925.72 *	422.50	1029.71	413.50
19	524.61	0.30	754.53 *	440.60	771.74 *	155.30	778.35 *	350.20	652.15 *	128.00	739.19	268.50
20	241.97	4.50	537.88	2904.50	524.81 *	1824.20	547.95 *	720.40	480.10 *	1184.20	522.68	1658.30
21	687.60	1.40	992.83 *	942.90	1121.84 *	759.10	978.82 *	1544.90	884.84 *	2556.20	994.58	1450.80
22	740.66	2.10	1036.11 *	1741.90	1052.98 *	1189.90	1045.91 *	673.50	950.79 *	254.90	1021.45	965.00
23	835.26 *	3391.30	1041.04 *	1226.90	1081.48 *	1288.20	1080.02 *	1523.10	950.09 *	1456.30	1038.16	1373.60
24	1026.60 *	53.30	1190.70	515.60	1083.14 *	796.30	1111.27 *	178.40	1046.63	430.80	1107.93	480.30
25	827.39	2.40	1419.42 *	3154.80	1374.68 *	2539.30	1405.65 *	2246.10	1180.57 *	3930.60	1345.08	2967.70
26	819.56	0.40	1285.01 *	2314.60	1344.66 *	2170.30	1405.57 *	2913.80	1234.39 *	1798.10	1317.41	2299.20
27	1082.65 *	486.50	1327.06 *	4162.10	1378.01 *	1343.60	1326.16 *	2643.80	1262.93 *	2717.00	1323.54	2716.60
28	1042.12	129.80	2587.23 *	4473.90	2629.38 *	5289.00	2654.75	5258.00	2368.88 *	5241.00	2560.06	5065.50
29	1162.96 *	549.60	2212.22 *	3025.50	2107.87 *	3895.10	2270.44	4406.90	2175.31	5187.10	2191.46	4128.60
30	1033.42 *	2165.90	1816.05 *	4969.20	1850.78 *	5126.60	1856.54 *	3936.30	1578.41	4982.50	1775.44	4753.70
31	1306.07 *	5096.10	2311.11 *	5207.10	2305.51 *	5107.40	2436.42 *	4538.80	2076.07 *	5099.30	2282.28	4988.20
32	1303.52 *	4492.40	2322.17	5083.20	2267.82 *	5255.00	2308.40 *	3908.20	2034.68 *	5356.00	2233.27	4900.60
33	1301.06 *	4842.10	2285.94 *	5000.40	2390.58 *	4853.60	2416.77 *	5388.00	2046.00 *	4713.70	2284.82	4988.90
34	713.51	3007.40	1212.04 *	5020.60	1237.27 *	5168.70	1235.58 *	5402.90	1079.61	5385.90	1191.13	5244.50
35	870.63	2616.50	1419.37	5315.50	1477.05 *	5165.80	1538.30 *	5291.00	1306.19	4289.70	1435.22	5015.50
36	592.87	5264.70	1782.99 *	4608.70	1834.97 *	5069.60	1728.69 *	4785.50	1572.49	5032.10	1729.79	4874.00
moyenne	770.77	892.09	1140.44	1718.15	1149.14	1596.47	1168.25	1558.33	1052.29	1687.56	1127.53	1640.13

TAB. 4.8 – Résultats pour le 2|UO|L-CVRP (non séquentiel et orienté) avec 1h30 de temps de calcul maximal et 10 réplications

	classe 1		classe 2		classe 3		classe 4		classe 5		classe 2-5	
instance	coût	cpu	coût	cpu	coût	cpu	coût	cpu	coût	cpu	coût	cpu
1	278.73	0.00	278.73	0.50	284.23	0.70	282.95	1.00	278.73	0.90	281.16	0.80
2	334.96	0.00	334.96	0.00	352.16	0.50	334.96 *	0.00	334.96	0.00	339.26	0.10
3	358.40	0.00	380.35	4.60	385.32 *	1.70	362.41	1.10	358.40	0.70	371.62	2.00
4	430.88	0.00	430.88	1.50	430.88	0.90	447.37	1.00	430.88	0.40	435.00	0.90
5	375.28	0.00	375.28	1.40	379.94	4.50	383.87 *	2.40	375.28	0.80	378.59	2.30
6	495.85	0.00	495.85	0.60	498.16	5.60	498.32	1.90	495.85	0.00	497.04	2.00
7	568.56	0.00	715.02	5.00	674.23 *	0.90	702.45	8.20	658.64	1.50	687.58	3.90
8	568.56	0.00	665.17 *	2.50	738.43	119.00	692.47	17.00	621.85	3.00	679.48	35.40
9	607.65	0.00	607.65	0.40	607.65	1.80	625.10 *	13.70	607.65	0.30	612.01	4.00
10	535.80	0.00	683.89 *	46.80	615.68	2.60	710.87	43.10	690.96	279.30	675.35	93.00
11	505.01	0.00	671.54 *	84.80	699.35 *	37.80	773.75 *	40.70	636.77	26.90	695.35	47.60
12	610.00	0.20	610.00	47.00	610.00	14.90	614.23 *	6.90	610.23	15.20	611.12	21.00
13	2006.34	0.00	2502.65 *	42.60	2409.55 *	128.20	2577.99 *	63.20	2416.04	92.60	2476.56	81.70
14	837.67	0.20	1033.69	67.90	989.22 *	229.30	981.42 *	206.00	925.04	118.10	982.34	155.30
15	837.67	0.00	1006.84 *	82.90	1141.95 *	923.60	1165.07 *	92.80	1199.74	1098.30	1128.40	549.40
16	698.61	0.00	698.61	1.30	698.61	1.00	703.35	5.50	698.61	4.90	699.80	3.20
17	861.79	0.00	861.79 *	0.90	861.79 *	1.00	861.79	9.90	861.79	1.20	861.79	3.20
18	723.54	8.30	987.52 *	832.90	1022.57 *	635.10	1104.08 *	697.00	925.72	179.80	1009.97	586.20
19	524.61	0.30	721.32 *	5331.50	753.66	457.40	762.24 *	235.60	652.15	221.50	722.34	1561.50
20	241.97	4.50	490.85 *	3209.20	517.57 *	4107.40	539.37 *	2320.00	479.06	931.80	506.71	2642.10
21	687.60	1.40	965.02 *	3961.30	1089.20 *	876.90	972.13 *	5360.10	890.29	3037.80	979.16	3309.00
22	740.66	2.10	994.69	3569.30	1044.25 *	2154.30	1061.90	1752.10	945.49	3171.70	1011.58	2661.90
23	835.26 *	3391.30	992.51 *	2790.10	1064.00 *	1707.20	1063.96 *	4667.00	950.17	852.10	1017.66	2504.10
24	1026.60 *	53.30	1151.11 *	1175.80	1077.16 *	1108.70	1103.63	1422.30	1047.35 *	832.80	1094.81	1134.90
25	827.39	2.40	1360.60 *	4388.40	1355.40 *	2476.90	1394.87 *	3928.40	1186.93	3914.60	1324.45	3677.10
26	819.56	0.40	1257.93 *	5181.50	1321.19 *	4450.90	1392.17 *	1412.70	1228.69 *	1799.30	1299.99	3211.10
27	1082.65 *	486.50	1274.48 *	4033.10	1347.45 *	3382.80	1315.52 *	3480.80	1266.68	4005.40	1301.03	3725.50
28	1042.12	129.80	2524.02 *	5180.20	2584.88 *	5000.20	2625.12 *	5139.80	2352.10	5390.60	2521.53	5177.70
29	1162.96 *	549.60	2157.33 *	5100.70	2074.16 *	4302.90	2248.50 *	4210.90	2167.47	5280.10	2161.86	4723.60
30	1033.42 *	2165.90	1767.69 *	5250.80	1824.70	4688.60	1832.93 *	5323.10	1569.69	4034.00	1748.75	4824.10
31	1306.07 *	5096.10	2217.87 *	4523.80	2276.46 *	5333.70	2416.85	5069.70	2063.89	5145.10	2243.77	5018.10
32	1303.52 *	4492.40	2244.13	5331.90	2237.45 *	3802.60	2273.51 *	5357.10	2010.07	5392.90	2191.29	4971.10
33	1301.06 *	4842.10	2230.17	5425.50	2347.49 *	5239.10	2392.08 *	4938.70	2026.57	5305.10	2249.08	5227.10
34	713.51	3007.40	1166.18	5388.30	1202.75 *	5308.90	1217.89	5395.60	1071.24	5395.20	1164.52	5372.00
35	870.63	2616.50	1368.57	5156.10	1446.30	5350.30	1528.99	5377.80	1296.83	5404.50	1410.17	5322.20
36	592.87	5264.70	1732.71	5385.20	1803.00 *	4211.80	1713.83	5401.70	1565.99	5293.50	1703.88	5073.10
moyenne	770.77	892.09	1109.93	2266.84	1132.41	1835.27	1157.72	2000.13	1052.72	1867.55	1113.19	1992.45

TAB. 4.9 – Résultats pour le 2|UR|L-CVRP (non séquentiel et avec rotation) avec 1h30 de temps de calcul maximal et 10 réplications

Conclusion

Nous nous sommes intéressé, dans le cadre de cette thèse, à la modélisation et à la résolution de problèmes difficiles venant de l'ordonnancement et du transport pour finir par l'étude d'un problème mêlant transport et ordonnancement. Nous avons proposé des méthodes approchées dans le but de résoudre les problèmes de manière rapide et efficace. Nous nous sommes focalisé sur cinq problèmes particuliers qui font partie, soit des problèmes de tournées de véhicules, soit des problèmes d'ordonnancement et nous avons proposé des méthodes originales de résolution .

Dans un premier temps, nous nous sommes intéressé à des problèmes d'ordonnancement. Nous avons conçu un modèle et des algorithmes pour le RCPSP facilement adaptables afin de pouvoir y intégrer aisément des contraintes additionnelles. Nous avons donc modélisé ce problème à l'aide de multiflots et développé des méthodes qui tirent profit de cette modélisation. Cela nous a permis de considérer des extensions originales du RCPSP qui incluent des contraintes supplémentaires entre les ressources comme les time lags conditionnels. Il n'existe pas, à notre connaissance, de *benchmark* pour le RCPSP avec *time lags* conditionnels. La méthode que nous avons développée a été également testée sur des instances de RCPSP classique et a été classée parmi 28 méthodes de la littérature. Elle occupe le premier tiers du classement.

Nous avons ensuite proposé une approche originale pour un problème de placement orthogonal en deux dimensions (2OPP). Cette approche consiste à relaxer le problème en RCPSP, puis à se servir de la solution du RCPSP pour construire une solution au 2OPP associé. Nous avons mis en œuvre cette idée à travers deux méthodes différentes : la première repose sur un schéma classique de génération d'ordonnancements, la seconde utilise un multiflot. Les résultats expérimentaux montrent la pertinence de cette approche. Ils ont été comparés à quatre méthodes existantes : les résultats confirment que notre méthode est compétitive par rapport aux meilleurs méthodes publiées. De plus, elle propose des solutions dans des temps de calcul très faibles, nettement plus faibles que ceux proposés dans la littérature.

Ensuite, nous avons traité deux problèmes de tournées de véhicules très différents. Le premier est le *Stacker Crane Problem* préemptif et asymétrique. Nous avons montré qu'il est possible de transformer ce problème en un problème moins contraint de construction d'arbre. L'efficacité de cette approche a été mise en évidence en comparant les résultats avec ceux d'une méthode exacte. Le second problème est un problème de transport à la demande auquel nous avons ajouté une contrainte financière. Ce problème, très contraint, a été résolu grâce à une heuristique d'insertion et des techniques de propagation de contraintes.

Il n'existe pas de *benchmark* pour ce problème. Néanmoins, nous avons comparé notre méthode à deux méthodes de la littérature sur des instances sans contrainte financière. Les temps de calcul que nous obtenons sont largement inférieurs à ceux de ces deux méthodes. Toutefois, les fonctions objectifs étant différentes, il est difficile de comparer les coûts des tournées.

Enfin, le dernier problème abordé est le 2L-CVRP. Il mêle problèmes d'ordonnancement et de transport : il intègre des contraintes de chargement en deux dimensions à un problème classique de tournées de véhicules. Ainsi, un sous-problème de placement orthogonal en deux dimensions (2OPP) est à considérer lors de la résolution du 2L-CVRP. Les contraintes de chargement ont été résolues grâce aux résultats sur la relaxation du 2OPP que nous avons fourni dans le chapitre deux. Ceci nous a conduit à considérer un nouveau problème : le RCPSP-CVRP, qui est une relaxation du 2L-CVRP. Nous avons utilisé un schéma GRASP×ELS et tiré profit de la méthode *Split* en alternant entre l'espace des solutions et l'espace des tours géants. Cette approche s'est montrée particulièrement efficace : elle a été comparée à quatre méthodes de la littérature. Notre méthode fournit une diminution significative des coûts des tournées.

Cette thèse contient différentes originalités :
- plusieurs problèmes abordés sont nouveaux : le RCPSP avec time lags conditionnels et le problème de transport à la demande avec contraintes financières ;
- plusieurs méthodes proposées sont nouvelles : représentation des solutions d'un problème de *pickup and delivery* sous forme d'arbre, résolution grâce à un multiflot d'un problème de placement ;
- plusieurs relaxations de problèmes connus n'avait jamais été envisagées : relaxation d'un problème de placement en problème de RCPSP, relaxation du 2L-CVRP en RCPSP-CVRP.

Bien que nous abordons dans cette thèse uniquement des méthodes approchées, nous apportons des résultats exacts sur des questions de modélisation. En particulier, nous avons montré :
- la possibilité de représenter, sous forme d'arbres bipartis ordonnés, des tournées relatives à un problème de *pickup and delivery* avec préemption ;
- l'équivalence entre un multiflot possédant des propriétés de non circuit et une solution du 2OPP.

Finalement, durant cette thèse, nous avons été amené à nous intéresser :
- à des problèmes de modélisation qui reposent sur des outils venant de la théorie des flots et réseaux et/ou de la recherche opérationnelle en général ;
- à des problèmes d'optimisation et plus particulièrement à la conception de schémas algorithmiques exploitant les modélisations proposées. Ceci inclut des heuristiques de construction, dont le but est de proposer une solution initiale et des métaheuristiques (GRASP, VNS, GRASP×ELS, ...), qui agissent sur une solution et tentent de l'améliorer par des techniques de diversification (mutations, voisinage large) et d'intensification (recherche locale).

Ce travail ouvre plusieurs perspectives :

- l'application des multiflots à d'autres extensions du RCPSP ;
- l'application de la propagation de contraintes à un problème de transport à la demande avec correspondances ;
- la résolution du 3L-CVRP en considérant une relaxation du problème.

Notations

Notations pour le RCPSP (sections 2.2 et 2.3)

A	ensemble d'activités
n	nombre d'activités
d_i, $i \in A$	durée de l'activité i
R	ensemble de ressources
m	nombre de types de ressources
M_k, $k \in R$	quantité de ressources de type k disponible
r_{ik}, $i \in A$, $k \in R$	consommation de l'activité i en ressource k
$i \ll j$, $i \in A$, $j \in A$	contrainte de précédence entre i et j
Δ_i, $i \in A$	date de début (au plus tôt) de i
C_{max}	makespan (date de fin du projet)
s	source (activité fictive)
p	puits (activité fictive)
A^*	ensemble des activités réelles et fictives ($A^* = A \cup \{s, p\}$)
$F = (f_1 \dots f_m)$	multiflot représentant le transfert de ressources entre activités
$F_{(i,j)} = (f_{1(i,j)} \dots f_{m(i,j)})$	quantités de ressources transférées de i vers j
$E_{\ll}$	ensemble des arcs induits par une contrainte de précédence
E_F	ensemble des arcs qui portent une quantité non nulle de flot
$G = (X, E)$	graphe orienté représentant une solution du RCPSP
X	ensemble des sommets de G : $X = A^*$
E	ensemble des arcs de G : $E = E_{\ll} \cup E_F$

TAB. 4.10 – Principales notations pour le RCPSP (problème et solution)

G'	graphe partiel
(U, V)	coupe dans G'
λ	séquence des activités
x_0	activité à insérer dans le graphe partiel
$pred(i)$	ensemble des activités j telles que $(j, i) \in E$
$succ(i)$	ensemble des activités j telles que $(i, j) \in E$
$\pi(i)$	date de fin au plus tôt de i
$\bar{\pi}(i)$	longueur du plus long chemin entre i et p
$out(i)$	quantité de flot donnée à la coupe
$in(i)$	quantité de flot reçue de la coupe
$out_k(i)$, $k \in R$	quantité de flot donnée à la coupe pour la ressource k
$in_k(i)$, $k \in R$	quantité de flot reçue de la coupe pour la ressource k

TAB. 4.11 – Notations pour les algorithmes du RCPSP

$lag(i,j)$, $i \in A^*$, $j \in A^*$	time lag conditionnel entre i et j
Lag_t	l'ensemble des couples d'activités (i,j) tel que i transfère de la ressource à j à l'instant t
π_r	valeur de raccrochage

TAB. 4.12 – Notations additionnelles pour le RCPSP avec *time lags*

Notations pour le 2OPP (section 2.5 et 2.6)

B	ensemble d'objets
n	nombre d'objet
b_i, $i \in B$	un objet de B
l_i	largeur de l'objet b_i
L_i	longueur de l'objet b_i
C	conteneur
l_C	largeur du conteneur
L_C	longueur du conteneur
(x_i, y_i)	position du coin inférieur gauche de l'objet b_i dans le conteneur
$itMax1$	nombre maximal d'itérations pour la phase 1 de l'algorithme
$itMax2$	nombre maximal d'itérations pour la phase 2 de l'algorithme
$maxRepet = 5$	nombre maximal de répétitions de l'algorithme

TAB. 4.13 – Principales notations pour le 2OPP et le *double SGS*

$G = (X, E)$	graphe associé au *double flot* (X représente l'ensemble des objets et E est l'ensemble des arcs qui portent du flot)
X^*	ensemble X augmenté des sommets fictifs source et puits
$\mathcal{F}$	flot qui donne la position relative de gauche à droite
$\mathcal{G}$	flot qui donne la position relative de bas en haut
$E_{\mathcal{F}}$	ensemble des arcs orientés qui portent du flot $\mathcal{F}$
$E_{\mathcal{G}}$	ensemble des arcs orientés qui portent du flot $\mathcal{G}$
$E_{\mathcal{F}\cup\mathcal{G}}$	ensemble des arcs orientés de i vers j qui portent une quantité de flot $\mathcal{F}_{i,j} + \mathcal{G}_{i,j}$ non nulle
$E_{\mathcal{F}\cup\mathcal{G}^-}$	ensemble des arcs orientés de i vers j qui portent une quantité de flot $\mathcal{F}_{i,j} + \mathcal{G}_{j,i}$ non nulle
$G_x = (X^*, E_x)$	graphe qui sert à calculer les abscisses des objets
$G_y = (X^*, E_y)$	graphe qui sert à calculer les ordonnées des objets

TAB. 4.14 – Principales notations pour le *double flot*

Notations pour le SCP (section 3.2)

K	ensemble des demandes
o_k, $k \in K$	origine de la demande k
d_k, $k \in K$	destination de la demande k
$X = \{0, 1, \ldots, n\}$	ensemble de sommets ($0 =$ dépôt)
X_O	ensemble des nœuds origine
X_D	ensemble des nœuds destination
X_R	ensemble des nœuds relais
$d(x, x')$	distance entre x et x'
r	lien labellisé : $r = (x, y, k)$ représente le déplacement du véhicule de x vers y avec la charge de la demande k (si le véhicule est vide, alors $k = 0$)
$Origine(r)$	nœud de départ pour le lien labellisé r
$Dest(r)$	nœud d'arrivée pour le lien labellisé r
$Label(r)$	demande dont la charge est transportée pour le lien labellisé r (0 si le véhicule est vide)
Γ	tournée
$C_{tour}(\Gamma)$	coût de la tournée Γ
Γ^k	sous-tournée de Γ qui comprend uniquement des liens labellisés r tels que $Label(r) = k$
$\sigma_\Gamma(r)$	lien labellisé associé au lien r dans Γ : si $r = (x, o_k, 0)$, alors $\sigma_\Gamma(r) = (d_k, y, 0)$; si $r = (y, x, k)$ où x est un relais et k est non nul, alors $\sigma_\Gamma(r) = (x, z, k)$; $\sigma_\Gamma(r)$ est indéfini dans les autres cas
$Actif(\mathcal{A})$	ensemble des relais présents dans l'arbre $\mathcal{A}$
$Rel(\mathcal{A}, k)$	ensemble linéairement ordonné des relais fils de la demande k dans l'arbre $\mathcal{A}$
$Dem(\mathcal{A}, x)$	ensemble linéairement ordonné des demandes filles du relais x dans l'arbre $\mathcal{A}$
$C_{arbre}(\mathcal{A})$	coût de l'arbre $\mathcal{A}$
$Arbre(\Gamma)$	arbre associé à la tournée Γ

TAB. 4.15 – Principales notations pour le SCP

Notations pour le DARP (section 3.3)

n	nombre de demandes
$D = \{1, \ldots, n\}$	ensemble de demandes
K	nombre de véhicules / tournées
$X = \{0, 1, \ldots, 2n\}$	ensemble de sommets ($0 =$ dépôt)
$\delta(x, y)$, $x \in X$, $y \in X$	durée pour aller de x à y
$d(x, y)$, $x \in X$, $y \in X$	distance entre x et y
Q	capacité des véhicules (identiques pour tous les véhicules)
o_i, $i \in D$	origine de la demande i
d_i, $i \in D$	destination de la demande i
q_i, $i \in D$	charge (nombre de passagers) de la demande i
q_x, $x \in X$	charge associée au sommet x ($q_x = q_i$ si $x = o_i$, $q_x = -q_i$ si $x = d_i$)

$[e_x, l_x]$, $x \in X$	fenêtre de temps (contrainte initiale) associée au sommet x
$[a_x, b_x]$, $x \in X$	fenêtre de temps réduite (à cause des contraintes liées à la solution en construction) associée au sommet x
T	durée maximale d'une tournée
Δ_i, $i \in D$	durée maximale de transport pour la demande i
$t(x)$, $x \in X$	date de service au sommet x
$\Gamma = \{0, 1, \dots l+1\}$	une tournée (0 représente le dépôt pour le départ et $l+1$ le dépôt pour l'arrivée)
Γ_k, $k \in [1, \dots, K]$	la tournée associée au véhicule k
$dep_1(k)$, $k \in [1, \dots, K]$	sommet fictif représentant le dépôt de départ de la tournée Γ_k
$dep_2(k)$, $k \in [1, \dots, K]$	sommet fictif représentant le dépôt d'arrivée de la tournée Γ_k
$\Gamma_{[\gamma, \gamma_2]}$	sous tournée $\{x, \dots, y\}$ de $\Gamma = \{0, \dots, \gamma, \dots, \gamma_2, \dots, l+1\}$
tournée valide	tournée dans laquelle les contraintes de capacité et temporelles sont satisfaites
$x_{succ} \in \Gamma$	successeur de $x \in \Gamma$ dans la tournée Γ
$x_{pred} \in \Gamma$	prédécesseur de $x \in \Gamma$ dans la tournée Γ
D_Γ	l'ensemble des demandes transportées par le véhicule réalisant la tournée Γ
$Perf(\Gamma)$	critère à minimiser $Perf(\Gamma) = \alpha \times Durée(\Gamma) + \beta \times Ride(\Gamma) + \zeta \times Wait(\Gamma)$, α, β, $\zeta \in (\mathbb{R}^+)^3$
$charge(x)$, $x \in X$	nombre de passagers dans le véhicule après que le véhicule soit passé en x
λ_x, $x \in X$	coefficient du sommet x dans la fonction objectif
$\mathcal{L}_i$, $i \in D$, non insérée	liste des insertions (tournées et positions) possibles pour i
$\mathcal{LV}_i$, $i \in D$, non insérée	liste des tournées encore possibles pour insérer i

TAB. 4.16 – Principales notations pour le DARP

$c(x, y)$, $x \in X$, $y \in X$	coût financier pour aller de x à y (payable en $t(x)$)
R	capital initial
r_i $i \in D$	revenu associé à la demande i (payé en $t(d_i)$)
r_x $x \in X$	revenu associé au sommet x ($r_x = r_i$ si $x = d_i$, $r_x = 0$ sinon)
x_τ, $\tau \in [0, +\infty[$	le sommet x de plus grand $t(x)$ tel que $t(x) \leq \tau$
$\Gamma.Cash(x)$, $x \in X$	différence entre revenus et dépenses engendrés par la seule tournée Γ lorsque le véhicule associé à Γ part de x
$\Gamma.Cash(\tau)$, $\tau \in [0, +\infty[$	= différence entre revenus et dépenses engendrés par la seule tournée Γ à l'instant τ
$Total.Cash(\tau)$, $\tau \in [0, +\infty[$	balance financière à l'instant τ (différence entre le capital initial, l'ensemble des revenus et l'ensemble des dépenses)
$\mathcal{L}t$	liste utilisée pour traiter l'insertion d'une demande dans une tournée Γ_{k_0}, elle contient les sommets des tournées Γ_k, $k \neq k_0$ triés suivant leur date de service croissante

TAB. 4.17 – Notations pour les contraintes financières

Notations pour le 2L-CVRP (section 4)

n	nombre de clients (aussi égal au nombre de demandes)
Q	capacité (en terme de poids) des véhicules
K	nombre de véhicules disponibles
$\mathcal{A}$	aire de chargement d'un véhicule
L	longueur du véhicule
l	largeur du véhicule
m_i, $i = 1 \ldots n$	nombre de colis pour le client i
p_i, $i = 1 \ldots n$	poids total des colis à livrer au client i
L_{ik}, $k = 1 \ldots m_i$, i=$1 \ldots n$	longueur du colis k pour le client i
l_{ik}, $k = 1 \ldots m_i$, i=$1 \ldots n$	largeur du colis k pour le client i
$t = \{t_0, t_1, \ldots, t_{n(t)}, t_{n(t)+1}\}$	tournée composée des clients $t_1, \ldots, t_{n(t)}$, (t_0 et $t_{n(t)+1}$ représente le dépôt)
$n(t)$	nombre de clients dans la tournée t
$f(t)$	coût de la tournée t
N_{best}	nombre de solutions du RCPSP-CVRP sauvegardées durant le GRASP×ELS
P	valeur de la pénalité ajoutée au coût de la solution si le nombre de tournées est supérieur au nombre de véhicules disponibles
S	solution du RCPSP-CVRP
$t(S)$	ensemble des tournées associées à la solution S
$K(S)$	nombre de véhicules utilisés dans la solution S
$f(S)$	coût de la solution S
T	tour géant
N_{grasp}	nombre d'itérations du GRASP
N_{els}	nombre d'itérations de l'ELS
N_{mut}	nombre de mutations
N_{max}	nombre maximal d'itérations sans amélioration pour l'ELS
$T = \{T_1, \ldots, T_{n(T)}\}$	tour géant qui résulte de la concaténation des tournées T_1 à $T_{n(T)}$
$n(T)$	nombre de tournées dans le tour géant T
N_{map}	taille de la *map*
N_{min}	nombre minimal d'itérations de la procédure `test_RCPSP` pour sauvegarde dans la *map*
val	tableau stockant le résultat de `test_RCPSP`
N_1, N_2, N_3, N_4	nombre maximal d'itérations par étape pour `test_RCPSP`
N_{gen}	nombre maximal d'itérations dédiées à la génération d'une solution initiale
$d(i, j)$	distance de i à j
N_{RL}	nombre maximal d'itérations pour la recherche locale
$SEUIL$	seuil minimal d'amélioration de la recherche locale
α	coefficient réducteur de l'aire de chargement

TAB. 4.18 – Principales notations pour le 2L-CVRP

Index

2L-CVRP
- collision, 212
- map, 211
- RCPSP-CVRP, 195
- tournée valide, 192
- tournées RCPSP-équivalentes, 212

DARP
- fenêtre de temps réduite, 162
- propagation de contraintes, 162
- tournée valide, 161

Métaheuristique
- algorithme glouton, 23
- algorithme génétique, 36
- algorithme mémétique, 35
- algorithme évolutionnaire, 35
- bassin d'attraction, 23
- colonie de fourmis, 38
- descente, 26
- diversification, 23
- ELS, 33
- GRASP, 32
- GRASP×ELS, 34
- GRASP×ILS, 34
- ILS, 32
- intensification, 23
- optimum global, 23
- optimum local, 23
- path relinking, 37
- pénalité, 39
- random walk, 26
- recherche locale, 23, 26
- recherche tabou, 30
- recuit simulé, 29
- scatter search, 37
- transformation locale, 23, 26
- VNS, 30
- voisinage, 23, 26

Méthode exacte
- branch and bound, 42
- exploration arborescente, 40
- programmation linéaire, 46

RCPSP
- activité de raccrochage, 69
- coupe, 65
- diagramme de Gantt, 58
- flot, 63
- graphe, 56
- multiflot, 63
- puits, 63
- SGS, 59
- source, 63
- time lag conditionnel, 85
- time lag maximal, 84
- time lag minimal, 84
- valeur de raccrochage, 88

SCP
- arbre biparti ordonné, 140
- arbre consistant, 140
- arbre-reformulation, 139
- chemin fortement valide, 143
- condition de non chevauchement, 143
- lien labellisé, 134
- théorème de restriction, 136
- tournée fortement valide, 139
- tournée valide, 135

Bibliographie

[AEGS93] N. Ascheuer, L. Escudero, M. Grötschel, and M. Stoer. A cutting plane approach to the sequential ordering problem (with applications to job scheduling in manufacturing). *SIAM Journal on Optimization*, 3:25–42, 1993.

[AFGS10] C. Archetti, D. Feillet, M. Gendreau, and M. Grazia Speranza. Complexity of the VRP and SDVRP. *Transportation Research Part C*, 2010. doi:10.1016/j.trc.2009.12.006.

[AJR00] N. Ascheuer, M. Jünger, and G. Reinelt. A branch & cut algorithm for the asymmetric traveling salesman problem with precedence constraints. *Computational Optimization and Applications*, 17:61–84, 2000.

[AK88] M.J. Atallah and S. Rao Kosaraju. Efficient solutions to some transportation problems with applications to minimizing robot arm travel. *SIAM Journal on Computing*, 17:849–869, 1988.

[AM01] J. Alcaraz and C. Maroto. A robust genetic algorithm for resource allocation in project scheduling. *Annals of Operations Research*, 102:83–109, 2001.

[Ame00] W. Ben Ameur. Constrained length connectivity and survivable networks. *Networks*, 36(1):17–33, 2000.

[AMO93] R.V. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, 1993.

[AMOR95] R.K. Ahuja, T.L. Magnanti, J.B. Orlin, and M.R. Reddy. Applications of network optimization. In M.O. Ball, T.L. Magnanti, C.L. Monma, and G.L. Nemhauser, editors, *Network models, Handbooks in Operation Research and Management Science*, pages 1–83. Elsevier Science, 1995.

[AMR03] C. Artigues, P. Michelon, and S. Reusser. Insertion techniques for static and dynamic resource-constrained project scheduling. *European Journal of Operational Research*, 149:249–267, 2003.

[AMR04] J. Alcaraz, C. Maroto, and R. Ruiz. Improving the performance of genetic algorithms for the rcps problem. In *Proceedings of the ninth International Workshop on Project Management and Scheduling*, pages 40–43. 2004.

[AR00] C. Artigues and F. Roubellat. A polynomial activity insertion algorithm in a multi-resource schedule with cumulative constraints and multiple nodes. *European journal of operational research*, 127:297–316, 2000.

[ASA06] B. Abbasi, S. Shadrokh, and J. Arkat. Bi-objective resource-constrained project scheduling with robustness and makespan criteria. *Applied Mathematics and Computation*, 180:146–152, 2006.

[Ash06] D. Ashlock. *Evolutionary Computation for Modeling and Optimization.* Springer, 2006.

[AZ04] N. Azizi and S. Zolfaghari. Adaptive temperature control for simulated annealing: a comparative study. *Computers & Operations Research*, 31:2439–2451, 2004.

[Bap95] P. Baptiste. *Resource constraints for preemptive and non preemptive scheduling.* PhD thesis, University PARIS VI, 1995.

[BBK98] T. Baar, P. Brucker, and S. Knust. Tabu-search algorithms and lower bounds for the resource-constrained project scheduling problem. In S. Voss, S. Martello, I. Osman, and C. Roucairol, editors, *Meta-heuristics: Advances and trends in local search paradigms for optimization*, pages 1–8. Kluwer Academic Publishers, 1998.

[BBV08] R. Baldacci, M. Battarra, and D. Vigo. Routing a heterogeneous fleet of vehicles. In B. Golden, S. Raghavan, and E. Wasil, editors, *The Vehicle Routing Problem: Latest Advances and new Challenges*, pages 3–28. Springer, 2008.

[BC01] N. Beldiceanu and M. Carlsson. Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint. *Lecture notes in computer science*, 2239:377–391, 2001.

[BCGL07] G. Berbeglia, J.F. Cordeau, I. Gribkovskaia, and G. Laporte. Static pickup and delivery problems: a classification scheme and survey. *Journal of the Spanish Society of Statistics and Operations Research*, 15:1–31, 2007.

[BD04] P. Baptiste and S. Demassey. Tight lp bounds for resource constrained project scheduling. *OR Spectrum*, 26(2):251–262, 2004.

[BDM+99] P. Brucker, A. Drexl, R. Mohring, K. Neumann, and E. Pesch. Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research*, 112(1):3–41, 1999.

[Bea83] J.E. Beasley. Route-first cluster-second methods for vehicle routing. *Omega*, 11:403–408, 1983.

[BESW94] J. Blazewicz, K.H. Ecker, G. Schmidt, and J. Weglarz. *Scheduling in Computer and Manufacturing Systems.* Springer-Verlag, 2nd edition, 1994.

[BFP95] E. Balas, M. Fischetti, and W.R. Pulleyblank. The precedence constrained asymmetric traveling salesman problem. *Mathematical Programming*, 68:241–265, 1995.

[BK00] P. Brucker and S. Knust. A linear programming and constraint propoagation-based lower bound for the RCPSP. *European journal of operational research*, 127:355–362, 2000.

[BKKS98] J.W. Baugh, G. Krishna, R. Kakivaya, and J.R. Stone. Intractability of the dial-a-ride problem ans a multiobjective solution using simulated annealing. *Engineering optimization*, 30(2):91–123, 1998.

[BKST98] P. Brucker, S. Knust, A. Schoo, and O. Thiele. A branch and bound algorithm for the resource constrained project scheduling problem. *European journal of operational research*, 107(2):272–288, 1998.

[BL03a] K. Bouleimen and H. Lecocq. A new efficient simulated annealing algorithm for the resource-constrained project scheduling problem and its multiple mode version. *European journal of operational research*, 149(2):268–281, 2003.

[BL03b] K. Bouleimen and H. Lecocq. A new efficient simulated annealing algorithm for the resource-constrained project scheduling problem and its multiple modes version. *European Journal of Operational Research*, 149:268–281, 2003.

[BLLN06] P. Baptiste, P. Laborie, C. Lepape, and W. Nuijten. Constraint-based scheduling and planning. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 22, pages 761–799. Elsevier, 2006.

[Blu05] C. Blum. Ant colony optimization: Introduction and recent trends. *Physics of Life Reviews*, 2:353–373, 2005.

[BMM98] A. Balakrishnan, T.L. Magnanti, and P. Mirchandani. Designing hierarchical survivable networks. *Operations Research*, 46(1):116–136, 1998.

[BMR88] M. Bartusch, R.H. Möhring, and F.J. Radermacher. Scheduling project networks with resource constraints and time windows. *Annals of Operations Research*, 16(1):201–240, 1988.

[BR03] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3), 2003.

[BS05] M.J. Brusco and S. Stahl. *Branch-and-Bound Applications in Combinatorial Data Analysis*. Statistics and computing. Springer New York, 2005.

[CC88] J. Carlier and P. Chretienne. *Problèmes d'ordonnancement : modélisation, complexité, algorithmes*. Masson, 1988.

[CCM07] F. Clautiaux, J. Carlier, and A. Moukrim. A new exact method for the two-dimensional orthogonal packing problem. *European Journal of Operational Research*, 183(3):1196–1211, 2007.

[CDM92] A. Colorni, M. Dorigo, and V. Maniezzo. Distributed optimization by ant colonies. In *Proceedings of the 1rst European Conference on Artificial Life*, pages 134–142. Elsevier Publishing, 1992.

[Cer85] V. Cerny. Thermodynamical approach to the travelling salesman problem. *Journal of Optimization Theory and Applications*, 45(1):41–51, 1985.

[CGH96] I. Charon, A. Germa, and O. Hudry. *Méthodes d'optimisation combinatoire*. Masson, 1996.

[CH08] H. Chtourou and M. Haouari. A two-stage-priority rule based algorithm for robust resource-constrained project scheduling. *Computers and Industrial Engineering*, 55(1):183–194, 2008.

[CILSG10] J.F. Cordeau, M. Iori, G. Laporte, and J.J. Salazar-González. A branch-and-cut algorithm for the pickup and delivery traveling salesman problem with lifo loading. *Networks*, 55(1):46–59, 2010.

[CJCM08] F. Clautiaux, A. Jouglet, J. Carlier, and A. Moukrim. A new constraint programming approach for the orthogonal packing problem. *Computers & Operations Research*, 35:944–959, 2008.

[CL03] J.F. Cordeau and G. Laporte. A tabu search heuristic for the static multi-vehicle dial-a-ride problem. *Transportation Research Part B*, 37:579–594, 2003.

[CL07] J.F. Cordeau and G. Laporte. The dial-a-ride problem: models and algorithms. *Annals of Operations Research*, 153(1):29–46, 2007.

[CLP06] Feng Chu, Nacima Labadi, and Christian Prins. A scatter search for the periodic capacitated arc routing problem. *European Journal of Operational Research*, 169(2):586 – 605, 2006.

[CLSV07] J.F. Cordeau, G. Laporte, M.W.P. Savelsbergh, and D. Vigo. Vehicle routing. In C. Barnhart and G. Laporte, editors, *Handbook in Operations Research and Management Science*, pages 367–428. Elsevier, Amsterdam, 2007.

[CMC10] C.E. Cortés, M. Matamala, and C. Contardo. The pickup and delivery problem with transfers: Formulation and a branch-and-cut solution method. *European Journal of Operational Research*, 200:711–724, 2010.

[CN03] J. Carlier and E. Neron. On linear lower bounds for the resource constrained project scheduling problem. *European journal of operational research*, 149:314–324, 2003.

[CN07] J. Carlier and E. Neron. Computing redundant resources for the resource constrained project scheduling problem. *European Journal of Operational Research*, 176(3):1452–1463, 2007.

[Cor06] J.F. Cordeau. A branch-and-cut algorithm for the dial-a-ride problem. *Operations research*, 54(3):573–586, 2006.

[COS02] A. Cesta, A. Oddi, and S.F. Smith. A constraint-based method for project scheduling with time windows. *Journal of Heuristics*, 8(1):109–136, 2002.

[CS05] A.P. Chassiakos and S.P. Sakellaropoulos. Time-cost optimization of construction projects with generalized activity constraints. *Journal of Construction Engineering and Management*, 131(10):1115–1124, 2005.

[CT03] J. Coelho and L. Tavares. Comparative analysis of metaŰheuricstics for the resource constrained project scheduling problem. Technical report, Department of Civil Engineering, Instituto Superior Tecnico, Portugal, 2003.

[CW64] G. Clarke and J.W. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12(4):568–581, 1964.

[CW81] N. Chistophides and C.A. Whitlock. Network synthesis with connectivity constraints - a survey. In J.P. Brans, editor, *Operational Research'81*, pages 705–723. North-Holland Publishing company, 1981.

[Dam05] J. Damay. *Techniques de résolution basées sur la programmation linéaire pour l'ordonnancement de projet*. PhD thesis, Université Blaise Pascal - Clermont-Ferrand II, 2005.

[DD04] M. Diana and M.M. Dessouky. A new regret insertion heuristic for solving large-scale dial-a-ride problems with time windows. *Transportation Research Part B*, 38:539–557, 2004.

[DH97] E. Demeulemeester and W. Herroelen. New benchmark results for the resource-constrained project scheduling problem. *Management Science*, 43:1485–1492, 1997.

[Dje99] K. Djellab. Scheduling preemptive jobs with precedence constraints on parallel machines. *European journal of operational research*, 117(2):355–367, 1999.

[DLPP10] C. Duhamel, P. Lacomme, C. Prins, and C. Prodhon. A GRASP×ELS approach for the capacitated location-routing problem. *Computers & Operations Research*, 37(11 (à paraître)):1912–1923, 2010.

[DLQT09] C. Duhamel, P. Lacomme, A. Quilliot, and H. Toussaint. 2L-CVRP: a GRASP resolution scheme based on RCPSP. In *Computers & Industrial Engineering*, Troyes, 6-9 Juillet 2009. CIE 2009 International Conference on Computers & Industrial Engineering.

[DLQT10a] C. Duhamel, P. Lacomme, A. Quilliot, and H. Toussaint. Définition d'un schéma d'optimisation GRASP×ELS pour le 2L-CVRP avec rotations de boîtes. In *Actes ROADEF 2010, 11e congrès de la Société Française de Recherche Opérationnelle et d'Aide à la Décision*, Toulouse, 24-26 février 2010.

[DLQT10b] C. Duhamel, P. Lacomme, A. Quilliot, and H. Toussaint. A multi-start evolutionary local search for the two dimensional loading capacitated vehicle routing problem. *En révision pour Computers & Operations Research*, 2010.

[DM41] B. Dushnik and E.W. Miller. Partially ordered sets. *American Journal of Mathematics*, 63:600–610, 1941.

[DPH00] U. Dorndorf, E. Pesch, and T. Phan Huy. A time-oriented branch-and-bound algorithm for resource-constrained project scheduling with generalized precedence constraints. *Management Science*, 46(10):1365–1384, 2000.

[DPST03] J. Dréo, A. Pétrowski, P. Siarry, and E. Taillard. *Métaheuristiques pour l'optimisation difficile*. Eyrolles, 2003.

[DQS07] J. Damay, A. Quilliot, and E. Sanlaville. Linear programming based algorithms for preemptive and non preemptive RCPSP. *European Journal of Operational Research*, 182(3):1012–1022, 2007.

[DRCL10] I. Dumitrescu, S. Ropke, J.F. Cordeau, and G. Laporte. The traveling salesman problem with pickup and delivery: polyhedral results and a branch-and-cut algorithm. *Mathematical programming*, 121(2):269–305, 2010.

[DRLV06] D. Debels, B. De Reyck, R. Leus, and M. Vanhoucke. A hybrid scatter search/electromagnetism meta-heuristic for project scheduling. *European Journal of Operational Research*, 169(2):638–653, 2006.

[DS94] G. Dahl and M. Stoer. A polyhedral approach to multicommodity survivable network design. *Numerische Mathematik*, 68:149–167, 1994.

[Fag74] R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. *SIAM-AMS Proceedings*, 7:43–73, 1974.

[FDHI07] G. Fuellerer, K.F. Doerner, R.F. Hartl, and M. Iori. Ant colony optimization for the two-dimensional loading vehicle routing problem: detailed results. Technical report, University of Vienna, 2007. téléchargeable à http://prolog.univie.ac.at/research/VRPandBPP/.

[FDHI09] G. Fuellerer, K.F. Doerner, R.F. Hartl, and M. Iori. Ant colony optimization for the two-dimensional loading vehicle routing problem. *Computers & Operations Research*, 36:655–673, 2009.

[FG65] D.R. Fulkerson and J.R. Gross. Incidence matrices and interval graphs. *Pacific Journal of Mathematics*, 15:835–855, 1965.

[FG92] G.N. Frederickson and D.J. Guan. Preemptive ensemble motion planning on a tree. *SIAM Journal on Computing*, 21:1130–1152, 1992.

[FG93] G.N. Frederickson and D.J. Guan. Nonpreemptive ensemble motion planning on a tree. *Journal of Algorithms*, 15(1):29–60, 1993.

[FH97] P. Fortemps and M. Hapke. On the disjunctive graph for project scheduling. *Journal Foundations of Computing and Decision Sciences*, 22:195–209, 1997.

[FHK78] G.N. Frederickson, M.S. Hecht, and C.E. Kim. Approximation algorithms for some routing problems. *SIAM Journal on Computing*, 7:178–193, 1978.

[Fle93] G. Fleury. *Méthodes stochastiques et déterministes pour les problèmes NP-difficiles.* PhD thesis, Université Blaise Pascal - Clermont-Ferrand, 1993.

[FR89] T.A. Feo and M.G.C. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations research letters*, 8(2):67–71, 1989.

[Fra57] A.S. Fraser. Simulation of genetic systems by automatic digital computers. *Australian Journal of Biological Sciences*, 10:484–491, 1957.

[FSdV07] S.P. Fekete, J. Schepers, and J.C. Van der Veen. An exact algorithm for higher-dimensional orthogonal packing. *Operations research*, 55(3):569–587, 2007.

[GD00] L.M. Gambardella and M. Dorigo. An ant colony system hybridized with a new local search for the sequential ordering problem. *INFORMS Journal on Computing*, 12(3):237–255, 2000.

[GDB83] B.L. Golden, J.S. Dearmon, and E.K. Baker. Computational experiments with algorithms for a class of routing problems. *Computers & Operations Research*, 10(1):47–59, 1983.

[GILM06] M. Gendreau, M. Iori, G. Laporte, and S. Martello. A tabu search algorithm for a routing and container loading problem. *Transportation Science*, 40(3):342–350, 2006.

[GILM08a] M. Gendreau, M. Iori, G. Laporte, and S. Martello. A tabu search heuristic for the vehicle routing problem with two-dimensional loading constraints. *Networks*, 51(1):4–18, 2008.

[GILM08b] M. Gendreau, M. Iori, G. Laporte, and S. Martello. A tabu search heuristic for the vehicle routing problem with two-dimensional loading constraints. *Networks*, 51(1):4–18, 2008.

[GJ79] M. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

[GLM00] F. Glover, M. Laguna, and R. Martí. Fundamentals of scatter search and path relinking. *Control and Cybernetics*, 29(3):653–684, 2000.

[Glo77] F. Glover. Heuristics for integer programming using surrogate constraints. *Decision sciences*, 8(1):156–166, 1977.

[Glo86] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 13:533–549, 1986.

[Gon07] J.F. Goncalves. A hybrid genetic algorithm-heuristic for a two-dimensional orthogonal packing problem. *European Journal of Operational Research*, 183(3):1212–1229, 2007.

[GP06] L. Gouveia and P. Pesneau. On extended formulations for the precedence constrained asymmetric traveling salesman problem. *Networks*, 48(2):77–89, 2006.

[Har98] S. Hartmann. A competitive genetic algorithm for resource-constrained project scheduling. *Naval Research Logistics*, 45:733–750, 1998.

[Har99] S. Hartmann. Project scheduling models. In *Project Scheduling under Limited Resources : Models, Methods, and Applications*, number 478 in Lecture Notes in Economics and Mathematical Systems, chapter 2, pages 5–31. Springer-Verlag Berlin and Heidelberg GmbH & Co. K, 1999.

[Har00] S. Hartmann. Packing problems and project scheduling models : an integrating perspective. *The Journal of the Operational Research Society*, 51(9):1083–1092, 2000.

[Har02] S. Hartmann. A self-adapting genetic algorithm for project scheduling under resource constraints. *Naval Research Logistics*, 49:433–448, 2002.

[HBss] S. Hartmann and D. Briskorn. A survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*, in press. doi:10.1016/j.ejor.2009.11.005.

[HDR99] W. Herroelen, E. Demeulemeester, and B. De Reyck. A classification scheme for project scheduling. In J. Weglarz, editor, *Project Scheduling: Recent Models, Algorithms and Applications*, chapter 1, pages 1–26. Kluwer Academic, 1999.

[Her05] W. Herroelen. Project scheduling-theory and practice. *Productions and Operations Management*, 14(4):413–432, 2005.

[HG03] M. Haouari and A. Gharbi. An improved max-flow based lower bound for minimizing maximum lateness on identical parallele machines. *Operations Research Letters*, 31(1):49–52, 2003.

[HM01] P. Hansen and N. Mladenovic. Variable neighborhood search: Principles and applications. *European Journal of Operational Research*, 130:449–467, 2001.

[HPSG09] H. Hernández-Pérez and J. Salazar-González. The multicommodity one-to-one pickup-and-delivery traveling salesman problem. *European Journal of Operational Research*, 196(3):987–995, 2009.

[ISGV07] M. Iori, J.J. Salazar-González, and D. Vigo. An exact approach for the vehicle routing problem with two-dimensional loading constraints. *Transportation Science*, 41(2):253–264, 2007.

[JGH+05] J.F.Cordeau, M. Gendreau, A. Hertz, G. Laporte, and J.S. Sormany. New heuristics for the vehicle routing problem. In A. Langevin and D. Riopel, editors, *Logistics Systems: Design and Optimization*, chapter 9, pages 279–297. Springer, US, 2005.

[JLB07] R.M. Jorgensen, J. Larsen, and K.B. Bergvinsdottir. Solving the dial-a-ride problem using genetic algorithms. *Journal of the Operational Research Society*, 58(10):1321–1331, 2007.

[JMRW04] J. Josefowska, M. Mika, R. Rozycki, and G. Waligora. An almost optimal heuristic for preemptive cmax scheduling of dependant tasks on parallel identical machines. *Annals of Operations Research*, 129:205–216, 2004.

[JOPW86] J.J. Jaw, A.R. Odoni, H.N. Psaraftis, and N.H.M. Wilson. A heuristic algorithm for the multi-vehicle advance request dial-a-ride problem with time windows. *Transportation research. Part B : methodological*, 20(3):243–257, 1986.

[KD96] R. Kolisch and A. Drexl. Adaptive search for solving hard project scheduling problems. *Naval Research Logistics*, 43:23–40, 1996.

[KGV83] S. Kirkpatrick, S. Gelatt, and M. Vecchi. Optimization by simualted annealing. *Science*, 220(4598):671–680, 1983.

[KH99] R. Kolisch and S. Hartmann. Heuristic algorithms for solving the resource-constrained project scheduling problem: classification and computational analysis. In J. Weglarz, editor, *Project scheduling: Recent models, algorithms and applications*, pages 147–178. Kluwer academic, Amsterdam, the Netherlands, 1999.

[KH06] R. Kolisch and S. Hartmann. Experimental investigation of heuristics for resource-constrained project scheduling: an update. *European Journal of Operational Research*, 174:23–37, 2006.

[Kim01] A. Kimms. *Mathematical programming and financial objectives for scheduling projects.* Kluwer Academic, 2001.

[Kle00] R. Klein. Project scheduling with time-varying resource constraints. *International Journal of Production Research*, 38(16):3937–3952, 2000.

[KLQT10a] H. Kerivin, M. Lacroix, A. Quilliot, and H. Toussaint. Résolution heuristique du stacker crane problem préemptif et asymétrique à l'aide d'une arbre représentation des tournées. In *Actes ROADEF 2010, 11e congrès de la Société Française de Recherche Opérationnelle et d'Aide à la Décision*, Toulouse, 24-26 février 2010.

[KLQT10b] H. Kerivin, M. Lacroix, A. Quilliot, and H. Toussaint. Tree based heuristics for the preemptive asymmetric stacker crane problem. In *ISCO International Symposium on Combinatorial Optimization*, Hammamet, Tunisia, 24-26 mars 2010.

[KLQT10c] H. Kerivin, M. Lacroix, A. Quilliot, and H. Toussaint. Tree based models and algorithms for a specific pick up and delivery problem. *En révision pour RAIRO-Operations Research*, 2010.

[Kol95] R. Kolisch. *Project scheduling under resource constraints - Efficient heuristics for several problem classes.* Physica-Verlag, Heidelberg, 1995.

[Kol96a] R. Kolisch. Efficient priority rules for the resource-constrained project scheduling problem. *Journal of operations management*, 14:179–192, 1996.

[Kol96b] R. Kolisch. Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation. *European Journal of Operational Research*, 90:320–333, 1996.

[Kol00] R. Kolisch. Integrated scheduling, assembly area- and part-assignment for large-scale, make-to-order assemblies. *International Journal of Production Economics*, 64(1):127–141, 2000.

[KP01] R. Kolisch and R. Padman. An integrated survey of deterministic project scheduling. *Omega*, 29:249–272, 2001.

[KS99a] R. Klein and A. Scholl. Computing lower bounds by destructive improvement: An application to resource-constrained project scheduling. *European journal of operational research*, 112(2):322–346, 1999.

[KS99b] R. Klein and A. Scholl. Computing lower bounds by destructive improvement: An application to resource-constrained project scheduling. *European Journal of Operational Research*, 112:322–346, 1999.

[KS00] R. Klein and A. Scholl. Rogress: Optimally solving the generalized resource constrained project scheduling problem. *Mathematical Methods of Operations Research*, 52(3):467–488, 2000.

[KS03] Y. Kochetov and A. Stolyar. Evolutionary local search with variable neighborhood for the resource constrained project scheduling problem. In *Proceedings of the Third International Workshop of Computer Science and Information Technologies*. Russia, 2003.

[KSD95] R. Kolisch, A. Sprecher, and A. Drexl. Characterization and generation of a general class of resource constrained project scheduling problems. *Management Science*, 41(10):1693–1703, 1995.

[Lac09] M. Lacroix. *Le problème de ramassage et livraison préemptif : complexité, modèles et polyèdres.* PhD thesis, Université Blaise Pascal - Clermont-Ferrand II, 2009.

[LCT03] T.W. Leung, C.K. Chan, and M.D. Troutt. Application of a mixed simulated annealing-genetic algorithm heuristic for the two-dimensional orthogonal packing problem. *European Journal of Operational Research*, 145(3):530–542, 2003.

[LLRKS93] E.L. Lawler, K.J. Lenstra, A.H.G. Rinnoy-Kan, and D.B. Schmoys. Sequencing and scheduling: Algorithms and complexity. In S.C. Graves, A.H.G. Rinnoy-Kan, and P.H. Zipkin, editors, *Handbooks in Operation Research and Management Sciences*, chapter 9, pages 445–521. Elsevier Science, 1993.

[LM99] M. Laguna and R. Marti. GRASP and path relinking for 2-layer straight line crossing minimization. *INFORMS Journal on Computing*, 11(1):44–52, 1999.

[LMS03] H.R. Lourenço, O.C. Martin, and T. Stützle. Iterated local search. In F. Glover and G.A. Kochenberger, editors, *Handbook of Metaheuristics*, pages 321–353. Kluwer Academic Publishers, USA, 2003.

[LPRC04] P. Lacomme, C. Prins, and W. Ramdane-Cherif. Competitive memetic algorithms for arc routing problems. *Annals of Operations Research*, 131:159–185, 2004.

[LR95] V.J. Leon and B. Ramamoorthy. Strength and adaptability of problem-space based neighborhoods for resource-constrained scheduling. *OR Spektrum*, 17:173–182, 1995.

[LS07] Y. Luo and P. Schonfeld. A rejected-reinsertion heuristic for the static dial-a-ride problem. *Transportation Research Part B*, 41:736–755, 2007.

[LW08] S.S. Liu and C.J. Wang. Resource-constrained construction project scheduling model for profit maximization considering cash flow. *Automation in Construction*, 17(8):966–974, 2008.

[LZZZ10] S.C.H. Leung, X. Zhou, D. Zhang, and J. Zheng. Extended guided tabu search and a new packing algorithm for the two-dimensional loading vehicle routing problem. *Computers & Operations Research*, in press, 2010. doi:10.1016/j.cor.2010.04.013.

[MB03] M.A. Manier and C. Bloch. A classification for hoist scheduling problems. *The international Journal of Flexible Manufacturing Systems*, 15(12):37–55, 2003.

[MC70] R.R. Muntz and E.G. Coffman. Preemptive scheduling of real-time tasks on multiprocessor systems. *Journal of the ACM*, 17(2):324–338, 1970.

[MF90] P.B. Mirchandani and R.L. Francis. *Discrete Location Theory*. JOHN WILEY and SONS, 1990.

[MML06] S. Mitrovic-Minic and G. Laporte. The pickup and delivery problem with time windows and transshipment. *INFOR*, 44:217–227, 2006.

[MMRB98] A. Mingozzi, V. Maniezzo, S. Ricciardelli, and L. Bianco. An exact algorithm for the resource-constrained project scheduling problem based on a new mathematical formulation. *Management Science*, 44(5):714–729, 1998.

[MMS02] D. Merkle, M. Middendorf, and H. Schmeck. Ant colony optimization for resource constrained project scheduling. *IEEE Transactions on Evolutionary Computation*, 6:333–346, 2002.

[MQ97] A. Moukrim and A. Quilliot. A relation between multiprocessor scheduling and linear programming. *Order*, 14(3):269–278, 1997.

[MQ05] A. Moukrim and A. Quilliot. Optimal preemptive scheduling on a fixed number of identical parallel machines. *Operation Research Letters*, 33(2):143–150, 2005.

[MSG08] R. Montemanni, D.H. Smith, and L.M. Gambardella. A heuristic manipulation technique for the sequential ordering problem. *Computers & Operations Research*, 35:3931–3944, 2008.

[MV98] S. Martello and D. Vigo. Exact solution of the two dimensional finite bin packing problem. *Management Sciences*, 44(3):388–399, 1998.

[NI02] K. Nonobe and T. Ibaraki. Formulation and tabu search algorithm for the resource constrained project scheduling problem. In C.C. Ribeiro and P. Hansen, editors, *Essays and surveys in metaheuristics*, pages 557–588. Kluwer Academic Publishers, Dordrecht, 2002.

[NSZ06] K. Neumann, C. Schwindt, and J. Zimmermann. Resource-constrained project scheduling with time windows. In *Perspectives in Modern Project Scheduling*, number 92 in International Series in Operations Research & Management Science, chapter 15, pages 375–407. Springer US, 2006.

[Pat84] J.H. Patterson. A comparizon of exact approaches for solving the multiple constrained resource project scheduling problem. *Management Sciences*, 30(7):854–867, 1984.

[PCF08] C. Prins, R. Wolfler Calvo, and A. El Fallahi. A memetic algorithm and a tabu search for the multi-compartment vehicle routing problem. *Computers and Operations Research*, 35:1725–1741, 2008.

[PCL09] J. Paquette, J.F. Cordeau, and G. Laporte. Quality of service in dial-a-ride operations. *Computers & Industrial Engineering*, 56(1):1721–1734, 2009.

[PD98] P.M. Pardalos and D.Z. Du. *Network Flows: Theory, Algorithms, and Applications*, volume 40 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1998.

[PDH08a] S.N. Parragh, K.F. Doerner, and R.F. Hartl. A survey on pickup and delivery problems - part I: Transportation between customers and depot. *Journal für Betriebswirtschaft*, 58(1):21–51, 2008.

[PDH08b] S.N. Parragh, K.F. Doerner, and R.F. Hartl. A survey on pickup and delivery problems - part II: Transportation between pickup and delivery locations. *Journal für Betriebswirtschaft*, 58(2):81–117, 2008.

[PDH10] S.N. Parragh, K.F. Doerner, and R.F. Hartl. Variable neighborhood search for the dial-a-ride problem. *Computers & Operations Research*, 37:1129–1138, 2010.

[Pri04] C. Prins. A simple and effective evolutionary algorithm for the vehicle routing problem. *Computers & Operations Research*, 31(12):1985–2002, 2004.

[Pri09a] C. Prins. A GRASP×evolutionary local search hybrid for the vehicle routing problem. In Francisco Baptista Pereira and Jorge Tavares, editors, *Bio-inspired Algorithms for the Vehicle Routing Problem*, 161, pages 35–53. Springer, Berlin / Heidelberg, 2009.

[Pri09b] C. Prins. Two memetic algorithms for heterogeneous fleet vehicle routing problem. *Engineering Applications of Artificial Intelligence*, 22:916–928, 2009.

[QT10a] A. Quilliot and H. Toussaint. About casting 2D-bin packing into network flow theory. Technical report, Laboratoire LIMOS, Université Clermont-Ferrand II, 2010. Téléchargeable à http://www.isima.fr/limos/publi/RR-10-11.pdf.

[QT10b] A. Quilliot and H. Toussaint. Prospective network flow models and algorithms for bin packing problems. In *ISCO International Symposium on Combinatorial Optimization*, Hammamet, Tunisia, 24-26 mars 2010.

[RBL02] J. Renaud, F. Boctor, and G. Laporte. Perturbation heuristics for the pickup and delivery traveling salesman problem. *Computers & Operations Research*, 29(9):1129–1141, 2002.

[RH98] B. De Reyck and W. Herroelen. A branch and bound procedure for the resource-constrained project scheduling problem with generalized precedence relations. *European Journal of Operational Research*, 111(1):152–174, 1998.

[Sch00] A. Schirmer. Case-based reasoning and improved adaptive search for project scheduling. *Naval Research Logistics*, 47:201–222, 2000.

[SJB+93] W.M. Spears, K.A. De Jong, T. Bäck, D.B. Fogel, and H. de Garis. An overview of evolutionary computation. In P.B. Brazdil, editor, *Proceedings of the European Conference on Machine Learning (ECML-93)*, 667, pages 442–459, Vienna, Austria, 1993. Springer Verlag.

[SS97] S. Salhi and M. Sari. A multi-level composite heuristic for the multi-depot vehicle fleet mix problem. *European Journal of Operational Research*, 103(1):95–112, 1997.

[ST00] C. Schwindt and N. Trautmann. Batch scheduling in process industries: An application of resource-constrained project scheduling. *OR Spectrum*, 22(4):501–524, 2000.

[TL00] P. Tormos and A. Lova. Project scheduling with time varying resource constraints. *International journal of production research*, 38(16):3937–3952, 2000.

[TL01] P. Tormos and A. Lova. A competitive heuristic solution technique for resource-constrained project scheduling. *Annals of Operations Research*, 102:65–81, 2001.

[TL03a] P. Tormos and A. Lova. An efficient multi-pass heuristic for project scheduling with constrained resources. *International journal of production research*, 41:1071–1086, 2003.

[TL03b] P. Tormos and A. Lova. Integrating heuristics for resource constrained project scheduling: One step forward. Technical report, Department of Statistics and Operations Research, University of Valencia, 2003.

[TP92] M.T. Fiala Timlin and W.R. Pulleyblank. Precedence constrained routing and helicopter scheduling: heuristic design. *Interfaces*, 22(3):100–111, 1992.

[Tur36] A. Turing. On computable numbers, with an application to the Entscheidungs problem. *Proceedings of London Mathematical Society*, 42:544–546, 1936. Available online at http://www.abelard.org/turpap2/tp2-ie.asp.

[TV97] P. Toth and D. Vigo. Heuristic algorithms for the handicapped persons transportation problem. *Transportation science*, 31(1):60–71, 1997.

[TV02] P. Toth and D. Vigo. An overview of vehile routing problems. In P. Toth and D. Vigo, editors, *The Vehicle Routing Problem*, pages 1–26. SIAM Monographs on Discrete Mathematics and Application, Philadelphia, 2002.

[Van06] M. Vanhoucke. Work continuity constraints in project scheduling. *Journal of Construction Engineering and Management*, 132(1):14–25, 2006.

[VBQ03] V. Valls, F. Ballestin, and S. Quintanilla. A hybrid genetic algorithm for the RCPSP. Technical report, Department of Statistics and Operations Research, University of Valencia, 2003.

[VBQ05] V. Valls, F. Ballestin, and S. Quintanilla. Justification and RCPSP: A technique that pays. *European Journal of Operational Research*, 165(2):375–386, 2005.

[VMP+09] J.G. Villegas, J. Medaglia, C. Prins, C. Prodhon, and N. Velasco. GRASP/evolutionary local search hybrids for a truck and trailer routing problem. *VIII Metaheuristic International Conference, Hamburg, 13-16 Juillet*, 2009.

[VR09] A. Vasan and K. Srinivasa Raju. Comparative analysis of simulated annealing, simulated quenching and genetic algorithms for optimal reservoir operation. *Applied Soft Computing*, 9:274–281, 2009.

[WM07] Steffen Wolf and Peter Merz. Evolutionary local search for the super-peer selection problem and the p-hub median problem. In T. Bartz-Beielstein et al., editor, *Hybrid Metaheuristics*, number 4771 in Lecture Notes in Computer Science, pages 1–15. Springer-Verlag, Berlin / Heidelberg, 2007.

[XCC06] Z. Xiang, C. Chu, and H. Chen. A fast heuristic for solving a large-scale static dial-a-ride problem under complex constraints. *European journal of operational research*, 174(2):1117–1139, 2006.

[ZTK09] E.E. Zachariadis, C.D. Tarantilis, and C.T. Kiranoudis. A guided tabu search for the vehicle routing problem with two-dimensional loading constraints. *European Journal of Operational Research*, 195:729–743, 2009.