Université de Montréal

Une heuristique à grand voisinage pour un problème de confection de tournée pour un seul véhicule avec cueillettes et livraisons et contrainte de chargement

par
Jean-François Côté

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des arts et des sciences
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Avril, 2009

Université de Montréal
Faculté des arts et des sciences

Ce mémoire intitulé:

Une heuristique à grand voisinage pour un problème de confection de tournée pour un seul véhicule avec cueillettes et livraisons et contrainte de chargement

présenté par:

Jean-François Côté

a été évalué par un jury composé des personnes suivantes:

Bernard Gendron,	président-rapporteur
Jean-Yves Potvin,	directeur de recherche
Michel Gendreau,	codirecteur
Jacques Ferland,	membre du jury

RÉSUMÉ

Dans ce mémoire, nous présentons un nouveau type de problème de confection de tournée pour un seul véhicule avec cueillettes et livraisons et contrainte de chargement. Cette variante est motivée par des problèmes similaires rapportés dans la littérature. Le véhicule en question contient plusieurs piles où des colis de hauteurs différentes sont empilés durant leur transport. La hauteur totale des items contenus dans chacune des piles ne peut dépasser une certaine hauteur maximale. Aucun déplacement n'est permis lors de la livraison d'un colis, ce qui signifie que le colis doit être sur le dessus d'une pile au moment d'être livré. De plus, tout colis i ramassé avant un colis j et contenu dans la même pile doit être livré après j. Une heuristique à grand voisinage, basé sur des travaux récents dans le domaine, est proposée comme méthode de résolution. Des résultats numériques sont rapportés pour plusieurs instances classiques ainsi que pour de nouvelles instances.

Mots clés: Tournée de véhicules, cueillettes et livraisons, contrainte de chargement, voyageur de commerce.

ABSTRACT

In this work, we consider a new type of pickup and delivery routing problem with last-in-first-out loading constraints for a single vehicle with multiple stacks. This problem is motivated by similar problems reported in the literature. In the problem considered, items are collected and put on top of one of multiple stacks inside the vehicle, such that the total height of the items on each stack does not exceed a given threshold. The loading constraints state that if items i and j are in the same stack and item i is collected before item j, then i must be delivered after j. Furthermore, an item can be delivered only if it is on the top of a stack. An adaptive large neighborhood heuristic, based on recent studies in this field, is proposed to solve the problem. Numerical results are reported on many classical instances reported in the literature and also on some new ones.

Keywords: Vehicule routing problem, pickup and delivery, loading constraint, last-in-first-out.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

REMERCIEMENTS

Tout d'abord, je désire remercier mes parents Ginette et Christian. Ils ont toujours cru en moi et ils m'ont toujours encouragé à aller plus loin. Sans eux, sans leur générosité et leur amour je ne me serais probablement pas rendu où je suis. Je suis heureux de vous avoir comme parents et je vous aime.

Je remercie mes professeurs Jean-Yves Potvin et Michel Gendreau qui, tout au long de ces quelques années, m'ont épaulé, écouté et qui ont contributué grandement à ce travail. Leur patience, leur sagesse et leurs idées m'ont soutenu à accomplir une des plus grandes choses de ma vie.

Finalement, je veux remercier mes amis, qui sont d'un grand support. Spécialement Paul, Yoan, Vincent, David et Éloïse, qui ont toujours été là pour m'aider. Je les apprécie tous et je ne les changerais pour rien au monde.

CHAPITRE 1

INTRODUCTION

L'arrivée des ordinateurs a favorisé l'émergence de plusieurs domaines de recherche. La recherche opérationnelle en est un bon exemple. Introduite dans les années '40, cette discipline a pour but de profiter au maximum des ressources disponibles afin de réaliser une tâche le plus efficacement possible. Dès lors, des problèmes d'apparence simple, mais avec un très grand nombre de solutions, devinrent le centre d'intérêt de nombreux chercheurs. On pense au large éventail d'études portant sur le problème du voyageur de commerce (PVC) et de ses variantes. Dans ce probléme, un voyageur doit visiter un ensemble de lieux dispersés géographiquement de telle sorte que la distance totale parcourue soit minimisée. Le nombre de séquences de visites différentes pour un problème à n lieux est de $\frac{(n-1)!}{2}$ et de ce nombre, seulement une (ou quelques unes) est de distance minimale, d'où la difficulté de résolution. Un nombre incalculable de recherches furent effectuées depuis les années '40, sans pour autant permettre de trouver la méthode miracle pour résoudre ce problème en des temps raisonnables. Toutefois, des problèmes avec plusieurs centaines de lieux peuvent maintenant être résolus de façon routinière sur les ordinateurs modernes. Applegate et al. (2004) ont même réussi à résoudre un problème avec 24 978 sommets en 2004, mais il s'agit d'un cas d'exception pour lequel des facilités de calcul hors de l'ordinaire ont été requises. Afin de résoudre des problèmes rencontrés dans la pratique, il faut donc souvent se tourner vers les méthodes heuristiques. Ces dernières ne garantissent pas l'obtention d'une solution optimale, mais permettent d'obtenir une qualité de solution acceptable en des temps d'exécution beaucoup moindres. On peut donc croire que les recherches sur ces sujets se poursuivront encore longtemps.

Plusieurs autres scientifiques se sont intéressés à des problèmes similaires au PVC que l'on rencontre dans les opérations quotidiennes. Par exemple, un problème récurrent dans le domaine des transports est la confection de tournées pour une flotte de véhicules devant effectuer des livraisons. Cette situation fut abordée pour la première fois

de manière scientifique par Dantzig et Ramser (1959). Ici, un ensemble de clients doit être visité par une flotte de véhicules. Chaque client demande qu'une certaine quantité d'un produit lui soit livrée par un véhicule. Étant donné que les véhicules ne peuvent pas transporter une quantité infinie du produit, plusieurs véhicules sont nécessaires pour satisfaire la demande totale de la clientèle. L'objectif du problème consiste alors à minimiser la somme des distances parcourues par les véhicules.

Les problèmes rencontrés par les compagnies de transport dans la pratique sont toutefois beaucoup plus complexes. Souvent, les clients doivent être visités à l'intérieur d'une fenêtre de temps. Dans d'autres cas, la flotte se compose de plusieurs types de véhicules avec des coûts d'opérations différents pour chaque type. Avec le temps, de plus en plus de variantes sont apparues afin de modéliser les contraintes de complexité grandissante observées dans la réalité.

Ainsi, ces dernières années, une toute nouvelle classe de problèmes de tournées de véhicules a été abordée. Cette classe contraint le placement des items à l'intérieur des véhicules afin de faciliter la séquence des livraisons. L'objectif principal est d'éviter que le chauffeur ait à déplacer des items afin de livrer l'item courant. En effet, de telles manipulations augmentent les temps de livraison et donc les coûts.

Les recherches effectuées dans ce mémoire sont motivées principalement par la gestion du placement des items à l'intérieur des véhicules pour des problèmes de tournées mixtes où s'effectuent à la fois des cueillettes et des livraisons. Puisqu'il s'agit d'un nouveau problème, l'effort est mis sur la création d'algorithmes pour un seul véhicule. Les techniques de résolution pour plusieurs véhicules pourraient toutefois faire appel à certains opérateurs définis dans ce travail. Le problème consiste donc à confectionner une tournée pour un seul véhicule où celui-ci doit effectuer un ensemble de cueillettes et de livraisons. Ce véhicule est muni de plusieurs piles, chacune de capacité limitée, où les colis recueillis peuvent y être déposés. Un colis ne peut être livré que s'il se trouve sur le dessus de la pile. Comme pour les problèmes précédents, l'objectif consiste à minimiser la distance totale parcourue par les véhicules.

Ce travail aborde donc une toute nouvelle problématique qui permet de généraliser deux problèmes connus dans la littérature. Le premier provient de Carrabs, Cordeau et

Laporte (2007) où un seul véhicule doit effectuer une série de cueillettes et de livraisons dont l'ordre doit respecter la contrainte habituelle de type dernier entré, premier sorti. En fait, ce problème est un cas particulier de notre problème où le véhicule possède une seule pile de capacité infinie. Les auteurs nomment ce problème le "Traveling Salesman Problem with Pickup and Delivery and LIFO Loading" ou TSPPDL. Le deuxième problème a été traité par Petersen et Madsen (2007) et consiste à confectionner une première tournée de cueillettes seulement qui forment un certain nombre de piles à l'intérieur du véhicule. Une fois les cueillettes complétées, une deuxième tournée est conçue pour les livraisons. Ici aussi, la livraison d'un item ne peut s'effectuer que si cet item est sur le dessus de sa pile. Le nom donné à ce problème est le "Double Traveling Salesman problem with Multiple Stacks" ou DTSPMS.

Le nom pour la problématique traitée dans ce mémoire est "problème de confection de tournée pour un seul véhicule avec cueillettes et livraisons et contrainte de chargement" ou 1-PDMS. Plusieurs métaheuristiques étaient candidates pour s'attaquer à ce problème. Flexibilité et simplicité furent les principaux facteurs menant au choix de la méthode. Malheureusement, peu d'approches de résolution semblaient offrir de telles caractéristiques. Les méthodes exactes requièrent des temps d'exécution exorbitants alors que certaines métaheuristiques sont fort complexes. Toutefois, une méthode qui a attiré notre attention est la métaheuristique à grand voisinage adaptatif de Pisinger et Ropke (2007). Sa principale force réside dans la simplicité avec laquelle l'espace des solutions est exploré. En effet, les solutions voisines de la solution courante sont générées à l'aide d'opérateurs de retrait et d'insertion qui sont relativement simples à concevoir et implanter. Par ailleurs, la satisfaction des contraintes est relativement facile à vérifier lorsqu'on insère de nouvelles requêtes dans une solution.

L'algorithme fonctionne à partir d'une solution de départ, construite avec une heuristique d'insertion à moindre coût. L'algorithme à grand voisinage améliore ensuite cette solution durant un certain nombre d'itérations. À chaque itération, un opérateur d'insertion et un opérateur de retrait sont sélectionnés. L'opérateur de retrait enlève d'abord un nombre aléatoire de requêtes de la solution courante, puis ces requêtes sont réintroduites dans la solution à l'aide de l'opérateur d'insertion. La nouvelle solution générée devient

alors la nouvelle solution courante si elle satisfait un critère bien défini. La prochaine itération débute alors avec la solution résultante. Une fois l'algorithme terminé, la meilleure solution rencontrée est optimisée avec un algorithme de programmation dynamique qui produit une tournée optimale respectant l'ordre selon lequel les requêtes sont empilées et dépilées.

Dans la suite, le chapitre 2 présente d'abord la formulation mathématique de notre problème. Le chapitre 3 fait ensuite une revue des approches de résolution pour les problèmes de tournées de véhicules avec contraintes de chargement. Ce chapitre introduit également les algorithmes de type "Détruire et Recréer". Le chapitre 4 constitue le coeur du mémoire et décrit les méthodes de résolution que nous avons développées pour notre problème. Le chapitre 5 rapporte ensuite les résultats obtenus sur des problèmes décrits dans la littérature ainsi que sur de nouveaux problèmes tests. Finalement, une conclusion résume notre contribution et apporte de nouvelles idées pour des recherches futures.

CHAPITRE 2

FORMULATION MATHÉMATIQUE

Soit n le nombre de requêtes à desservir et soit $G = (N,A)$ où $N = \{0,1,...,2n\}$ est l'ensemble des sommets, incluant le dépôt 0, et A est l'ensemble des arcs. Les sous-ensembles de sommets de cueillette et de livraison sont notés $P = \{1,...,n\}$ et $D = \{n+1,...,2n\}$, respectivement. À chaque sommet de cueillette $i \in P$ est associé un sommet de livraison $n+i \in D$. La demande aux sommets de cueillette et de livraison est dénotée $d_i \geq 0$ et $d_{n+i} = -d_i$ (la demande au dépôt est $d_0 = 0$). Un coût de déplacement c_{ij} est aussi associé à chaque arc $(i, j) \in A$.

Le problème consiste à confectionner une tournée pour un seul véhicule afin de minimiser ses coûts de déplacement. Cette tournée doit visiter tous les sommets $i \in P \cup D$ exactement une fois et doit commencer et se terminer au dépôt. Le véhicule dispose d'un ensemble $K = \{1,...,|K|\}$ de piles (compartiments) pour charger des colis. La hauteur de chaque pile $k \in K$ ne peut dépasser $Q \geq 0$. Tout colis $i \in P$ se retrouve nécessairement sur le dessus de la pile $k \in K$ choisie au moment de sa cueillette. Par ailleurs, tout colis à décharger doit se trouver sur le dessus d'une pile. Il existe donc une contrainte de type "dernier entré, premier sorti" au niveau de la tournée qui impose que tout colis j se trouvant au dessus de tout colis i dans la même pile k soit livré en premier.

Le modèle mathématique est présenté dans la suite, en posant les variables de décision :

- x_{ij} vaut 1 si le sommet j est visité immédiatement après le sommet i, 0 sinon, $\forall i, j \in N$;
- y_{ik} vaut 1 si la demande du sommet i se trouve sur la pile k, 0 sinon , $\forall i \in P, \forall k \in K$;
- $u_i \geq 0,$ est la position du sommet i dans la tournée , $\forall i \in N,$ avec $u_0 = 0$;
- $S_{ik} \geq 0$ est la hauteur de la pile k après avoir visité le sommet i , $\forall i \in N, \forall k \in K$;

Nous avons :

$$\min \sum_{i \in N} \sum_{j \in N, j \neq i} c_{ij} x_{ij} \tag{2.1}$$

sujet à

$$\sum_{j \in N} x_{ij} = 1 \ \forall i \in N \tag{2.2}$$

$$\sum_{j \in N} x_{ji} = 1 \ \forall i \in N \tag{2.3}$$

$$\sum_{k \in K} y_{jk} = 1 \ \forall j \in P \tag{2.4}$$

$$u_j \geq u_i + 1 - 2n(1 - x_{ij}) \ \forall i \in N, \forall j \in P \cup D \tag{2.5}$$

$$u_{n+j} \geq u_j + 1 \ \forall j \in P \tag{2.6}$$

$$S_{jk} \geq S_{ik} + d_j y_{jk} - Q(1 - x_{ij}) \ \forall i \in N, \forall j \in P, \forall k \in K \tag{2.7}$$

$$S_{(n+j)k} \geq S_{ik} + d_{n+j} y_{jk} - Q(1 - x_{i(n+j)}) \ \forall i \in N, \forall j \in P, \forall k \in K \tag{2.8}$$

$$S_{(n+j)k} \leq S_{jk} - d_j y_{jk} + Q(1 - y_{jk}) \ \forall j \in P, \forall k \in K \tag{2.9}$$

$$1 \leq u_i \leq 2n \ \forall i \in P \cup D \tag{2.10}$$

$$u_0 = 0 \tag{2.11}$$

$$S_{0k} = 0 \ \forall k \in K \tag{2.12}$$

$$0 \leq S_{ik} \leq Q \; \forall i \in P \cup D, \forall k \in K \tag{2.13}$$

La fonction économique en (2.1) consiste à minimiser la somme des coûts de déplacement. Les contraintes (2.2) et (2.3) assurent que chaque sommet est visité exactement une fois. La contrainte (2.4) exige qu'un item i se trouve dans une seule pile k. La position de chaque sommet $i \in N$ à l'intérieur de la tournée est définie par la contrainte (2.5). Pour forcer une livraison $n+i \in D$ à se retrouver après la cueillette correspondante $i \in P$, on introduit la contrainte (2.6). Les contraintes (2.7) et (2.8) représentent l'état de la pile au fur et à mesure des cueillettes et livraisons. Il faut noter que S_{ik} doit augmenter d'une valeur d_i dans le cas d'une cueillette et diminuer d'une valeur d_i dans le cas d'une livraison. La contrainte (2.9) impose l'ordre "dernier entré, premier sorti" sur la tournée. Celle-ci exige que l'état de la pile avant la cueillette $i \in P$ soit le même après la livraison $n+i$. La contrainte (2.10) est une borne supérieure sur les valeurs des u_i. En effet, puisqu'il y a $2n$ sommets dans le problème, la valeur des variables u_i ne peut être supérieure à ce nombre. Dans le même ordre d'idée, puisque le dépôt est le premier sommet de la tournée, sa variable u_i doit être posée à zéro, ce qui est fait par la contrainte (2.11). De plus, la hauteur des piles est posée à zéro à la sortie du dépôt par la contrainte (2.12). Enfin, la contrainte (2.13) assure que la capacité des piles soit respectée.

Il faut noter que la même solution est représentée $|K|!$ fois dans ce modèle. Pour un véhicule à deux piles, par exemple, il est facile de voir que le contenu des deux piles peut être interchangé sans modifier la solution. Des contraintes sont donc ajoutées à ce modèle afin de briser la symmétrie, soit :

$$y_{11} = 1 \tag{2.14}$$

$$y_{ik} \leq \sum_{j=1}^{i-1} y_{j(k-1)} \text{ pour } k > i \; , \forall i \in P \tag{2.15}$$

La contrainte (2.14) force l'item 1 à se retrouver dans la pile 1. Ensuite, la contrainte (2.15) stipule qu'un colis i peut-être mis dans une pile k seulement si la pile d'indice

inférieur est utilisée.

Nous terminons ce chapitre en notant que le problème 1-PDMS est NP-difficile. En effet, le problème du voyageur de commerce, qui est NP-difficile, est un cas particulier du problème $1-PDMS$ en associant à chaque sommet un sommet de cueillette et un sommet de livraison situés au même endroit et en posant $d_i = 0, \forall i \in N$.

CHAPITRE 3

REVUE DE LITTÉRATURE

Des centaines, voire des milliers, d'articles ont été écrits sur les problèmes de tournées de véhicules. La version classique du problème se formule de la façon suivante : étant donné un ensemble de clients dispersés géographiquement avec chacun une demande (e.g., quantité d'un produit) et une flotte de véhicules de capacité donnée, il faut déterminer une séquence de clients pour chacun des véhicules de sorte que tous les clients soient servis et que la distance totale parcourue par la flotte de véhicules soit minimale. De nombreuses variantes se greffent à ce problème de base : fenêtres de temps, flotte hétérogène, cueillettes et livraisons, etc. La première section de cette revue aborde différents algorithmes permettant de résoudre ces problèmes dont la métaheuristique "Détruire et Recréer". Dans la seconde section, les problèmes de tournées de véhicules avec empaquetage sont abordés. La dernière section traite plus particulièrement du problème que nous étudions dans ce mémoire, soit le problème de génération d'une tournée pour un seul véhicule avec contraintes de chargement.

3.1 Métaheuristiques récentes

Les algorithmes de résolution pour les problèmes d'optimisation combinatoire atteignent de nouveaux sommets en termes de vitesse d'exécution, de qualité des solutions et de sophistication. Il en est de même pour les algorithmes qui résolvent les problèmes de tournées de véhicules. De riches revues de la littérature sur le sujet se retrouvent dans Cordeau et al. (2005) et Bräysy et Gendreau (2005a, 2005b), où l'on traite des dernières avancées en terme d'heuristiques pour les problèmes de tournées de véhicules avec fenêtres de temps.

Pour citer quelques travaux récents, mentionnons d'abord Cordeau, Laporte et Mercier (2001) où les auteurs abordent le problème de tournées de véhicules avec fenêtres de temps. Un mécanisme particulier dans la recherche tabou permet de visiter des solutions

non réalisables à l'aide des insertions GENI de Gendreau, Hertz et Laporte (1992). Tarantilis et Kiranoudis (2002), quant à eux, combinent des segments de routes provenant de bonnes solutions. Par la suite, ils appliquent un algorithme de recherche tabou avec plusieurs voisinages différents. Mester et Bräysy (2005) combinent des stratégies évolutionnaires (Rechenberg 1973) et une recherche locale guidée (Voudouris 1997) pour résoudre un problème de tournées avec fenêtres de temps. Leur algorithme itératif fonctionne à deux niveaux. Le premier niveau consiste à optimiser la solution à l'aide d'une recherche locale guidée. Cette recherche utilise plusieurs voisinages pour améliorer la solution courante comme le "Relocate" de Savelsberg (1992), les 1-échanges de Osman (1993) et le 2-opt* de Potvin et Rousseau (1995). Le deuxième niveau retire des clients et les réintègre dans la solution à la manière de Shaw (1998) (voir plus loin).

Ce travail est basé sur un type particulier de métaheuristique, dérivé du concept "Détruire et Recréer" (Ruin-and-Recreate). Ce concept fut introduit par Schrimpf et al. (2000) pour résoudre un problème de tournées de véhicules avec fenêtres de temps, en s'inspirant de l'algorithme de Shaw (1998). L'idée principale consiste à retirer des éléments d'une solution pour ensuite les réintroduire de manière à générer une nouvelle solution. On décide alors si la nouvelle solution devient la solution courante ou non. Le tout est imbriqué dans une stratégie de recherche globale (d'où l'appartenance à la classe des métaheuristiques). En termes concrets pour les problèmes de tournées de véhicules, les opérations consistent à retirer des clients des routes pour les réinsérer dans un ordre différent dans les mêmes routes ou dans d'autres routes. Les dernières recherches sur ce type d'algorithme tentent de définir des opérateurs de retrait et d'insertion plus efficaces permettant d'atteindre des solutions de meilleure qualité. Dans Schrimpf et al. (2000), les auteurs définissent deux opérateurs de retrait et un opérateur d'insertion. Le premier opérateur de retrait consiste à choisir un client i déjà servi de manière aléatoire et de retirer i ainsi que ses q-1 plus proches clients, où q est choisi de manière aléatoire. L'autre opérateur de retrait retire tout simplement q clients de manière aléatoire. Pour l'opération d'insertion, les clients sont considérés dans un ordre arbitraire et sont réinsérés à l'endroit de moindre coût. La solution résultante est acceptée à l'aide du critère "Threshold Accepting", voir Dueck et Scheuer (1990). Ce critère permet d'accepter une

solution de qualité moindre tant que sa valeur ne dépasse pas un seuil prédéfini.

Dans Pisinger et Ropke (2007), les auteurs résolvent le problème de cueillettes et de livraisons avec fenêtres de temps à l'aide de cette approche. Dans ce travail, la méthode de Schrimpf et al. (2000) fut exploitée au maximum en définissant plusieurs opérateurs de retrait et d'insertion. Le mécanisme de sélection proportionnelle ou "roulette wheel selection" choisit les opérateurs de manière stochastique, selon une distribution de probabilités qui introduit un biais en faveur des opérateurs ayant eu le plus de succès. En plus, les auteurs expliquent comment transformer plusieurs variantes de problèmes de tournées en problèmes de cueillettes et livraisons.

Shaw (1998) utilise la programmation par contraintes lors de la réinsertion des clients pour des problèmes de tournées avec fenêtres de temps. Dell'Amico et al. (2007) résolvent le problème avec flotte hétérogène et fenêtres de temps en tentant, suite à la destruction partielle de la solution, de réaffecter les clients aux véhicules de manière à permettre l'introduction d'un nouveau type de véhicule.

L'algorithme rapporté dans ce mémoire se fonde essentiellement sur les travaux de Pisinger et Ropke (2007).

3.2 Tournées de véhicules avec empaquetage

Dans le passé, il était difficile de s'attaquer à la gestion du compartiment arrière des camions de livraison. Aujourd'hui, cette gestion devient possible avec les progrès de la technologie, ce qui permet de mieux modéliser la réalité. Dans ces problèmes, les véhicules possèdent des compartiments de chargement dans lesquels des items sont placés afin de les transporter à leurs points de livraison. La façon dont les items et les compartiments sont définis crée ainsi une nouvelle classe de problèmes de plus grande complexité.

Une des premières publications dans ce domaine est celle de Gendreau et al. (2008) qui traitent le problème de livraison de boîtes rectangulaires en deux dimensions à l'aide de camions ayant un compartiment de chargement rectangulaire, toujours en deux dimensions. Le problème de tournées est résolu par un algorithme de recherche tabou

similaire à celui rapporté dans Gendreau, Hertz et Laporte (1994) où la contrainte de capacité des véhicules peut être violée au cours de la recherche. En ce qui concerne le problème du placement des items, il est résolu à l'aide d'une métaheuristique exploitant l'heuristique "Bottom-Left" et la méthode exacte de Iori, Salazar-González et Vigo (2007). De plus, les auteurs considèrent deux règles pour placer les items, soit selon l'ordre des livraisons et sans ordre particulier. Finalement, les auteurs considèrent le même problème, mais en trois dimensions, dans Gendreau et al. (2006).

Un problème similaire est celui des tournées de véhicules "multi-piles" traité dans Doerner et al. (2007). Les camions possèdent ici des piles de même largeur et hauteur sur lequel des items peuvent être empilés. Les items sont des boîtes rectangulaires en deux dimensions et les largeurs autorisées dépendent de la largeur des piles. Par exemple, si la largeur d'une pile est de 5 centimètres et que le camion possède 4 piles, seuls des items de 5, 10, 15 et 20 centimètres de largeur pourront y être entreposés. Pour résoudre le problème de placement, les auteurs disposent d'une heuristique rapide et d'un algorithme exact de programmation dynamique. La recherche tabou de Gendreau et al. (2005) et les colonies de fourmis de Reimann, Stummer et Doerner (2002) sont utilisés pour résoudre le problème de tournées. Il faut noter que la largeur des items dans l'article de Gendreau et al. (2005) peut être discrétisée de manière à le transformer en un problème "multi-piles".

3.3 Tournées avec cueillettes et livraisons et contrainte de chargement

Cette section traite des problèmes de tournées de véhicules avec cueillettes et livraisons où une contrainte de chargement est imposée. Dans ces problèmes, chaque sommet de cueillette est associée à un sommet de livraison et la cueillette doit être effectuée avant la livraison.

Carrabs, Cordeau et Laporte (2007) traitent le problème de cueillette et livraison pour un seul véhicule avec une contrainte de type dernier entré, premier sorti. Ainsi, l'ordre des chargements et déchargements doit respecter cette contrainte. De façon plus précise, un colis i ramassé avant un colis j doit être livré après j. Les chercheurs définissent sept

opérateurs de voisinage intégrés au sein d'une métaheuristique de Recherche à Voisinage Variable ("Variable Neighboorhood Search"), voir Mladenovic et Hansen (1997). Dans Carrabs, Cerruli et Cordeau (2008) et Cordeau et al. (2007), les auteurs s'attaquent au problème à l'aide de méthodes exactes.

Erdogan, Cordeau et Laporte (2008) considèrent le même problème que Carrabs, Cordeau et Laporte (2007), mais avec la contrainte "premier entré, premier sorti". Cette contrainte se retrouve dans le transport de véhicules par traversier. Les véhicules entrent par l'arrière du bateau et en ressortent par l'avant. Leurs algorithmes consistent en une recherche tabou probabiliste et une recherche locale itérative.

Un problème plus particulier abordé par Petersen et Madsen (2007) concerne la confection d'une tournée de cueillettes afin de remplir un conteneur possédant plusieurs piles. Le conteneur est retourné au dépôt une fois que toutes les cueillettes ont été faites. Du dépôt, il est déplacé vers un autre dépôt dans le but d'effectuer les livraisons. Une autre tournée doit donc être réalisée. Cependant, celle-ci doit respecter l'ordre "dernier entré, premier sorti" des cueillettes. Dans cette étude, les auteurs définissent quatre algorithmes afin de résoudre le problème, soit un recuit simulé, une recherche tabou, une descente locale itérée et une recherche à grand voisinage.

CHAPITRE 4

MÉTHODE DE RÉSOLUTION

Dans ce travail, nous tenons compte à la fois de la confection de la tournée et du placement des items dans les piles. Pour ce faire, l'algorithme présenté tente de repositionner des requêtes en considérant l'état de la tournée et des piles. Le déplacement d'une requête consiste d'abord à retirer celle-ci de la solution. Par la suite, la requête est réintroduite dans la solution de manière à minimiser le coût additionnel de la tournée, soit en la plaçant dans une autre pile, soit en la plaçant dans la même pile mais à un autre endroit, tout en s'assurant de la réalisabilité. À cet effet, une structure de données a été conçue afin que l'algorithme d'insertion des requêtes dans la tournée ne considère que les positions qui satisfont les contraintes de chargement et les contraintes de capacité.

On pourrait toutefois considérer d'autres façons d'aborder le problème. Par exemple, il serait possible d'explorer le placement des items dans les piles et, pour chaque placement, générer la meilleure séquence possible de cueillettes et livraisons. Le problème de construction d'une tournée de moindre coût serait donc abordé de façon indirecte, ce qui n'est probablement pas l'idéal. Néanmoins, une telle approche est à la base de l'un des algorithmes présentés dans ce travail où une tournée optimale est générée à la toute fin en se basant sur le placement final des items dans les piles.

En choisissant de privilégier la résolution du problème de cueillettes et de livraisons, on a l'avantage de profiter des nombreux travaux sur ce sujet dans la littérature. Il faut toutefois tenir compte des contraintes liées au placement des items dans les piles lorsque l'on modifie les tournées, de façon à assurer la réalisabilité de la solution.

La métaheuristique choisie pour ce travail se base sur les travaux de Pisinger et Ropke (2007). Ceux-ci ont appliqué une approche de type "Détruire et recréer" pour résoudre des problèmes de tournées de véhicules avec cueillettes et livraisons et fenêtres de temps. Leur algorithme s'appelle "Adaptive Large Neighboorhood Search", ce qui peut se traduire par Recherche Adaptative à Grand Voisinage. En général, les métaheuristiques classiques ne font que de petites modifications locales à la solution courante, empêchant

ainsi la recherche d'explorer une large fraction de l'espace des solutions. L'idée ici est donc de compenser cette faiblesse en permettant des mouvements qui nous entraînent loin de la solution courante. Cette stratégie se base sur les travaux de Shaw (1998) où on peut potentiellement modifier de 30 à 40% d'une solution à chaque itération. Pour ce faire, les auteurs définissent des opérateurs de retrait et d'insertion. Un opérateur de retrait n'est qu'une simple heuristique qui retire des éléments de la solution courante selon un critère donné. Un opérateur d'insertion est par la suite appliqué pour remettre les éléments retirés dans la solution. Cette nouvelle solution peut ensuite être acceptée ou refusée. Les sections suivantes donneront donc des détails sur les opérateurs de retrait et d'insertion, le mécanisme d'acceptation et la manière de choisir les opérateurs à chaque itération.

4.1 Procédure de résolution

La procédure de résolution est constituée de trois algorithmes. Le premier algorithme s'occupe de créer la solution initiale. Le deuxième, l'algorithme à grand voisinage, est de nature itérative et se charge de trouver la meilleure solution possible à partir de la solution initiale. Un algorithme de programmation dynamique est exécuté à la fin pour identifier une tournée optimale associée au placement final des items dans les piles. Ces trois algorithmes sont exécutés l'un à la suite de l'autre.

4.2 Création de la solution initiale

Une première solution est nécessaire afin de lancer l'algorithme à grand voisinage. Toutefois, il y a peu d'intérêt à développer une solution initiale de très grande qualité car l'algorithme à grand voisinage peut en détruire une grande partie dès les premières itérations.

L'heuristique classique d'insertion à moindre coût a été choisie à cet effet. En premier lieu, les sommets sont ajoutés à une liste (dans un ordre aléatoire) à partir de laquelle l'algorithme procède à l'insertion itérative des sommets dans la tournée. Chaque requête est insérée à la position qui minimise le détour. La structure de données développée dans

ce travail permet d'obtenir toutes les positions respectant la contrainte de chargement des piles de type "dernier entré, premier sorti".

Il faut noter qu'il existe toujours une solution réalisable au problème en autant que la demande à un client ne dépasse pas la capacité des piles. En effet, une requête peut toujours être insérée à la fin de la tournée au moment où les piles sont vides puisqu'aucune contrainte sur la longueur de la tournée n'est imposée.

4.3 Algorithme à grand voisinage

L'algorithme à grand voisinage s'exécute selon le pseudo-code ci-dessous. À chaque itération, un opérateur d'insertion et un opérateur de retrait sont choisis, ainsi qu'un nombre q de requêtes à retirer et à réinsérer. L'opérateur de retrait se charge d'abord de retirer les q requêtes choisies. Celles-ci se retrouvent dans une liste qui est donnée à l'opérateur d'insertion afin de reconstruire la solution. Un critère d'acceptation prédéfini permet ensuite d'accepter ou non la nouvelle solution. Cette procédure est répétée pour un certain nombre d'itérations.

0. Procédure AlgorithmeàGrandVoisinage(Solution s, Entier nbIterations).

1. $iter \leftarrow 0$; $s^* \leftarrow s$;

2. Tant que $iter < nbIterations$

   2.1 Choisir un nombre aléatoire $q < n$ de requêtes ;

   2.2 Choisir un opérateur de retrait et d'insertion ;

   2.3 Appliquer les opérateur de retrait et d'insertion à s pour obtenir $s^{'}$;

   2.4 Poser $s \leftarrow s^{'}$ selon un certain critère d'acceptation ;

   2.5 Si $f(s^{'}) < f(s^*)$

      2.5.1 $s^* \leftarrow s^{'}$;

   2.6 $iter \leftarrow iter + 1$.

3. Retourner s^*.

Les sections suivantes décrivent les principaux éléments de l'algorithme ci-dessus.

4.4 Opérateurs de retrait

Les opérateurs de retrait détruisent une partie de la solution en y retirant certains éléments. Dans le problème considéré, les éléments à retirer sont les requêtes avec leur point de cueillette et de livraison. Lorsqu'une requête est retirée, la tournée demeure réalisable puisque la hauteur des piles est réduite et la contrainte de chargement demeure nécessairement satisfaite.

À la suite de l'application de la procédure de retrait, les portions de la solution ayant été détruites se retrouvent dans une liste de requêtes qui seront remises dans la solution avec un opérateur d'insertion.

4.4.1 Retrait aléatoire

Cet opérateur retire q requêtes de manière aléatoire. À chaque itération, les requêtes qui restent dans la solution sont considérées pour le retrait suivant avec une probabilité qui est la même pour toutes les requêtes. Les requêtes qui ont été retirées se retrouvent dans une liste qui est retournée à la fin de la procédure.

L'implantation de cet opérateur peut se faire en $O(n)$ en générant q nombres aléatoires différents dans l'intervalle [1, n]. À chaque itération, la requête dont l'indice de l'un des sommets correspond au nombre aléatoire généré est retirée.

4.4.2 Retrait selon la distance

Le retrait selon la distance est une forme particulière de l'opérateur de retrait de Shaw (1998) qui considère différentes métriques de corrélation entre deux requêtes (distance, fenêtres de temps, etc.). Voici le pseudo-code de l'algorithme de retrait selon la distance, où les fonctions $p(i)$ et $d(i)$ retournent les sommets de cueillette et de livraison, respectivement, de la requête i.

0. Procédure RetraitDistance(Solution s, Entier k, réel d)

1. $i \leftarrow$ ChoisirRequeteAleatoire(s) ;
2. $L \leftarrow \{i\}$;
3. $s \leftarrow s \setminus \{p(i), d(i)\}$;
4. Tant que $\mid L \mid < q$
   - 4.1 Choisir une requête i aléatoirement dans L
   - 4.2 $B \leftarrow \emptyset$
   - 4.3 Pour chaque requête $j \in s$
     - 4.3.1 $b_j \leftarrow c_{p(i)p(j)} + c_{d(i)d(j)}$;
     - 4.3.2 $B \leftarrow B \cup \{j\}$;
   - 4.4 Trier B selon les valeurs b_j ;
   - 4.5 Choisir un nombre aléatoire r dans l'intervalle $[0, 1]$
   - 4.6 $pos \leftarrow |B| \cdot r^d$;
   - 4.7 Sélectionner la requête j à la position pos dans B ;
   - 4.8 $L \leftarrow L \cup \{j\}$;
   - 4.9 $s \leftarrow s \setminus \{p(j), d(j)\}$;
5. Retourner L.

Pour débuter la procédure, une requête est choisie aléatoirement, retirée de la solution et mise dans une liste L. Une requête i est ensuite choisie aléatoirement dans cette liste à chacune des itérations subséquentes. Toutes les requêtes j qui sont encore dans la tournée sont triées en fonction d'un coût b_j, tel que défini à l'étape 4.3.1, qui correspond à la somme des distances entre les sommets de cueillette et de livraison des requêtes i et j. Une requête est ensuite choisie à l'aide d'une procédure de sélection probabiliste biaisée vers celles qui sont de plus petit coût b_j. En effet, avec une valeur élevée de l'exposant d, le terme r^d tend vers zéro, ce qui force l'algorithme à choisir des requêtes proches de i. À l'opposé, une valeur faible pour d produit un comportement se rapprochant de celui de l'opérateur de retrait aléatoire. Pour de bons résultats, Shaw (1998) suggère des

valeurs pour d dans l'intervalle $[5,20]$. Dans notre cas, la valeur de d fut posée à 6, suite à des expérimentations préliminaires.

4.4.3 Retrait orienté vers la tournée

Une des particularités de notre problème est qu'il n'y a qu'une seule tournée et il peut être intéressant de retirer une séquence de sommets consécutifs de cette tournée. Par ailleurs, toutes les requêtes ayant une cueillette (resp. livraison) à l'intérieur de la séquence choisie verront leur livraison (resp. cueillette) correspondante retirée également de la solution. Voici le pseudo-code de l'algorithme où les fonctions $p(i)$ et $d(i)$ retournent les sommets de cueillette et de livraison, respectivement, de la requête i. Par ailleurs, la fonction $prec(i)$ retourne la requête dont le sommet de cueillette ou de livraison précède immédiatement le sommet de cueillette de la requête i (la fonction retourne 0 si le sommet précécesseur est le dépôt). De façon similaire, la fonction $suiv(i)$ retourne la requête dont le sommet de cueillette ou de livraison suit immédiatement le sommet de cueillette de la requête i (la fonction retourne 0 si le sommet successeur est le dépôt).

0. Procédure RetraitPartieDeRoute(Solution s, Entier k)

    1. $i \leftarrow$ ChoisirRequêteAléatoire(s) ;
    2. $L \leftarrow \emptyset$;
    3. Tant que $|L| < q-1$
        - 3.1 $j \leftarrow prec(i)$;
        - 3.2 Si $j \neq 0$
            - 3.2.1 $L \leftarrow L \cup \{j\}$;
            - 3.2.2 $s \leftarrow s \setminus \{p(j), d(j)\}$;
        - 3.3 $j \leftarrow suiv(i)$;
        - 3.4 Si $j \neq 0$
            - 3.4.1 $L \leftarrow L \cup \{j\}$;
            - 3.4.2 $s \leftarrow s \setminus \{p(j), d(j)\}$,

4. $L \leftarrow L \cup \{i\}$;
5. $s \leftarrow s \setminus \{p(i), d(i)\}$;
6. Retourner L.

Au départ, l'algorithme choisit aléatoirement une requête i dont le sommet cueillette agit comme point d'ancrage. Ensuite, les $q-1$ requêtes autour du point d'ancrage sont retirées. Ainsi, l'algorithme considère le sommet j_1 précédant le sommet cueillette de i à l'aide la fonction $prec(i)$. Si j_1 n'est pas le dépôt, la requête associée à j_1 est retirée. Par la suite, le sommet j_2, successeur immédiat du sommet cueillette de i, est considéré à l'aide de la fonction $suiv(i)$. Si celui-ci n'est pas le dépôt, la requête associée à j_2 est retirée. Lorsque $q-1$ requêtes sont retirées, la requête i est retirée. À chaque itération, la requête qui est retirée de la solution est ajoutée à une liste L.

4.4.4 Retrait orienté vers les piles

L'opérateur de retrait orienté vers les piles retire les items en fonction de leur proximité dans les piles. En fait, cet opérateur est similaire à l'opérateur de retrait en fonction de la distance mais fait appel à une autre fonction de distance. Celle-ci s'exprime comme $b_j \leftarrow |pos(i) - pos(j)|$ où $pos(i)$ est la position du sommet i dans la pile (voir étape 4.3.1 de l'algorithme présenté à la section 4.4.2).

4.4.5 Choix du nombre de requêtes à retirer

Tel que mentionné précédemment, le nombre de requêtes à retirer doit être choisi à chaque itération. Si ce nombre est trop petit, il ne sera pas possible d'explorer largement le voisinage et la recherche risque de rester coincée dans des minima locaux de pauvre qualité. De plus, l'amplitude des améliorations possibles au coût de la solution risque d'être assez faible. Par ailleurs, si un nombre trop grand de requêtes est retiré, la reconstruction deviendra difficile et coûteuse et les solutions visitées seront probablement loin des minima locaux. Pisinger et Ropke (2007) remédient à cette difficulté en retirant un nombre aléatoire de requêtes. Cette manière de faire est intéressante car

elle permet une intensification de la recherche quand le nombre est faible et une diversification quand le nombre est grand. Ils ont d'abord suggéré un nombre q satisfaisant $4 \leq q \leq \min\{100, \alpha n\}$ où n est le nombre de requêtes et $\alpha = 0,4$ est une constante. Cependant, ils se sont aperçus que les solutions résultant du retrait d'environ 100 requêtes étaient rarement acceptées, car les heuristiques de reconstruction ne permettaient pas d'obtenir des solutions d'une qualité suffisante. Par ailleurs, les solutions créées à partir du retrait d'un petit nombre de requêtes n'apportaient aucune amélioration majeure. Ils ont donc effectué d'autres tests et ont suggéré d'utiliser un nombre q satisfaisant $\min\{0,1n;30\} \leq q \leq \min\{0,4n;60\}$.

Il n'est pas certain toutefois que ces formules s'appliquent à notre problème. Des tests intensifs furent donc effectués afin de mieux comprendre l'impact du nombre de requêtes retirées sur la qualité des solutions ainsi que sur le temps de calcul.

4.5 Opérateurs d'insertion

Les opérateurs d'insertion reconstruisent la solution à partir de la liste de requêtes retournée par l'opérateur de retrait. Comme il existe un nombre exponentiel de façons de réinsérer les requêtes dans la solution, il serait illusoire de vouloir identifier la suite des insertions menant à la meilleure solution (bien que Shaw (1998) propose un modèle de programmation mixte en nombres entiers, soutenu par la programmation par contraintes, qui permet d'obtenir l'ordre d'insertion optimal des requêtes). Les algorithmes proposés dans ce travail ont donc pour but d'obtenir une approche simple et rapide menant néanmoins à de bonnes solutions.

Pour chaque requête à insérer, les algorithmes d'insertion considèrent toutes les positions respectant la contrainte "dernier entré, premier sorti". Cette contrainte impose une structure particulière sur la route qui peut être exploitée.

Ainsi, en se basant sur la notation utilisée dans Carrabs, Cordeau et Laporte (2007), si une requête i est dans la pile k, le bloc $B_k(i, n+i)$ est la séquence de sommets du point de cueillette au point de livraison de la requête i. Ce bloc est dit *simple* s'il ne contient aucun sous-bloc. Il est *composé* autrement.

Deux requêtes i et j dans la pile k satisfont la contrainte "dernier entré, premier sortie" si les blocs $B_k(i,n+i)$ et $B_k(j,n+j)$ n'ont aucun sommet en commun ou si l'un des deux est un sous-bloc de l'autre. Autrement, ces blocs sont entrelacés et ne satisfont pas la contrainte "dernier entré, premier sorti".

Étant donné une position dans la route pour le sommet de cueillette de la requête i, un nombre réduit de positions respecteront la contraine "dernier entré, premier sorti" pour le point de livraison $n+i$. Une première possibilité consiste à insérer $n+i$ juste après i. En continuant à balayer la route, on a :

- si le sommet suivant n'est pas dans la pile k, alors on peut insérer $n+i$ après ce sommet.
- si le sommet suivant est dans la pile k et qu'il s'agit d'une cueillette j, alors toutes les positions d'insertion à l'intérieur du bloc $B_k(j,n+j)$ sont ignorées car les blocs $B_k(i,n+i)$ et $B_k(j,n+j)$ seraient alors entrelacés. La recherche d'une position réalisable se poursuit donc à partir de $n+j$.
- si le sommet suivant est dans la pile k et qu'il s'agit d'une livraison $n+j$, deux cas se présentent :
  - Si le sommet de cueillette j est situé après i dans la route, l'insertion de $n+i$ après $n+j$ est réalisable.
  - Si le sommet de cueillette j est situé avant i dans la route, il n'existe plus aucune position d'insertion réalisable dans la route, car les blocs $B_k(i,n+i)$ et $B_k(j,n+j)$ seraient alors entrelacés.

Les deux opérateurs d'insertion que nous avons utilisés sont décrits dans les sections suivantes.

4.5.1 Insertion à moindre coût

Cette heuristique insère les requêtes à l'endroit engendrant le détour minimal (i.e., qui minimise la distance additionnelle). Plus précisément, si la cueillette i est insérée

entre le sommet j et son successeur $suiv(j)$ et que la livraison $n+i$ est insérée entre l et $suiv(l)$ alors le détour est :

$$c_{j,i} + c_{i,suiv(j)} - c_{j,suiv(j)} + c_{l,n+i} + c_{n+i,suiv(l)} - c_{l,suiv(l)}$$

Les requêtes sont considérées une à une sans aucun ordre prédéfini et sont chacune insérées à l'endroit de moindre coût qui respecte la contrainte "dernier entré, premier sorti" ainsi que la contrainte de capacité.

4.5.2 Insertion avec regrets

Un désavantage de l'insertion à moindre coût se situe au niveau de la myopie de l'approche. En effet, en insérant une requête dans la solution, nous n'avons aucune idée de l'impact de celle-ci sur les prochaines requêtes à insérer. Il est malheureusement fort possible que l'ajout de cette requête fasse en sorte que des requêtes plus difficiles à placer soient insérées plus tard, au moment où il reste peu de d'alternatives réalisables. Un algorithme développé par Potvin et Rousseau (1995) tente de remédier à ce problème. L'approche consiste à insérer plus tôt les requêtes pour lesquelles le nombre de bonnes insertions réalisables est réduit (en effet, on risque d'éprouver du regret plus tard si on décide de ne pas les insérer tout de suite, d'où le nom d'algorithme d'insertion avec regrets). Il faut noter que Pisinger et Ropke (2007) utilisent aussi ce type d'insertion en considérant comme positions d'insertion possibles la meilleure position d'insertion dans chaque route. Puisqu'il n'y a qu'une seule route dans notre problème, l'implantation de cette approche considère plutôt le meilleur placement possible dans chaque pile. Plus précisément, en posant $|K| = m$, nous avons :

0. Procédure InsertionRegrets(Solution s, Entier m, Liste L)

1. Tant que $|L| > 0$:

    1.1 Pour chaque requête $i \in L$ faire

1.1.1 Pour chaque pile $k \in K$, calculer le détour minimum réalisable $\delta_{i,k}$ dans la route lorsqu'on place la requête i dans la pile k ;

1.1.2 Pour $j = 1,...,m$ poser $k_j \leftarrow$ indice de la pile avec le j^e plus petit détour ;

1.1.3 $regret_i \leftarrow \sum_{j=1}^{m} (\delta_{i,k_j} - \delta_{i,k_1})$;

1.2 $i^* \leftarrow \text{argmax}_{i \in L} \{regret_i\}$;

1.3 $L \leftarrow L \setminus \{i^*\}$;

1.4 Placer i^* dans la meilleure pile, soit celle entraînant le plus petit détour réalisable dans la tournée.

À partir de la liste L des requêtes à insérer, la procédure calcule pour chaque requête le détour minimum $\delta_{i,k}$ dans la route quand la requête est placée dans chaque pile $k \in K$. Par la suite, l'algorithme calcule le regret de chaque requête i. Plus le regret est grand, plus l'intérêt est grand de choisir la requête i au plus tôt et de la placer dans sa meilleure pile car sinon, il en coûtera cher pour la placer plus tard dans une autre pile.

4.6 Sélection d'un opérateur

Cette section décrit la façon dont les opérateurs sont choisis. La sélection des opérateurs définit en quelque sorte la manière dont l'algorithme explore l'espace des solutions. D'une part, le retrait aléatoire d'un grand nombre de requêtes, suivi d'une insertion à moindre coût, explore cet espace en largeur. D'autre part, le retrait d'un petit nombre de requêtes proches les unes des autres, couplé à une insertion avec regret, explore l'espace des solutions de manière circonscrite. De plus, certains opérateurs seront plus performants sur certains types de problèmes. Il importe donc de choisir un mécanisme intelligent de sélection des opérateurs pour mieux cerner ceux qui ont un grand impact sur la qualité des solutions.

Soit I l'ensemble des opérateurs de retrait, J l'ensemble des opérateurs d'insertion et un poids $w_i > 0$, $\forall i \in I \cup J$. Lorsque vient le moment de choisir un opérateur de retrait, le poids associé à chaque opérateur est calculé de la façon suivante :

$$\frac{w_i}{\sum_{j \in I} w_j} \; \forall i \in I$$

L'opérateur d'insertion est choisi de la même façon, de manière indépendante. Au départ, les poids w_i sont initialisés à 1. Puis, à chaque itération, les poids sont mis à jour afin de favoriser les opérateurs d'insertion et de retrait ayant eu le plus de succès. Les poids sont augmentés de la façon suivante :

1. Si la nouvelle solution est meilleure que la meilleure solution rencontrée jusque là, alors les poids des deux opérateurs sont augmentés d'une valeur σ_1 ;

2. Si la solution est moins bonne que la meilleure solution, mais est meilleure que la solution courante (tout en ayant satisfait le critère d'acceptation), alors les poids sont augmentés d'une valeur σ_2 ;

3. Si la solution est moins bonne que la solution courante mais a été acceptée, alors les poids sont augmentés d'une valeur σ_3.

Puisque les poids σ_i sont positifs, les poids des meilleures opérateurs risquent d'atteindre de très grandes valeurs rapidement et seront toujours sélectionnés par la suite. C'est pourquoi les poids sont réinitialisés à 1 à toutes les 500 itérations.

Un autre aspect à considérer est l'orientation de la recherche en fonction des poids choisis. Bien qu'il soit primordial de récompenser les opérateurs favorisant l'amélioration de la solution, il faut aussi diversifier la recherche afin d'atteindre les meilleures solutions possibles. Il faut donc noter qu'une très grande valeur pour σ_1 aura pour effet d'intensifier la recherche au détriment de la diversification. Les poids doivent donc être choisis judicieusement afin d'atteindre les meilleures solutions.

4.7 Critère d'acceptation des solutions

À chaque itération de l'algorithme, une nouvelle solution est générée à l'aide des différents opérateurs. L'algorithme d'acceptation pourrait considérer seulement les solutions qui améliorent la solution courante. Cependant, la diversification serait alors peu

favorisée et des minima locaux, possiblement de pauvre qualité, seraient rapidement atteints.

Dans ce travail, le mécanisme d'acceptation des nouvelles solutions est basé sur un processus similaire à celui que l'on retrouve dans le recuit simulé. Avec cette approche, lorsque la valeur $f(s')$ d'une nouvelle solution s' est meilleure que la valeur $f(s)$ de la solution courante, s' devient la nouvelle solution courante. Sinon, s' est acceptée avec une probabilité $e^{\frac{f(s)-f(s')}{T}}$ où s est la solution courante et $T > 0$ est un paramètre appelé température. Au départ, la température T est mise à $f(s)(1+z)$ où s est la solution de départ et z est un paramètre. Plus z est grand, plus la chance d'accepter des solutions de qualité moindre que la solution de départ est grande. À chaque itération, T devient $T \cdot c$ où $0 < c < 1$ est le taux de refroidissement. Puisque T diminue au fil des itérations, la probabilité d'accepter une solution de moindre qualité devient de moins en moins élevée.

La valeur de z fut posée égale à $0,05$ et celle de c à $0,99975$. Ces valeurs proviennent de l'article de Pisinger et Ropke (2007).

4.8 Algorithme d'optimisation par programmation dynamique

Afin d'optimiser la solution résultant de l'algorithme à grand voisinage, un algorithme de programmation dynamique génère la tournée optimale en fonction du placement des items sur les piles dans la solution finale.

Soit une séquence réalisable de chargements et déchargements décrivant l'évolution de chacune des piles. Par exemple, dans le cas où nous avons deux piles, soit $[1^+, 1^-, 2^+, 2^-]$ et $[3^+, 4^+, 4^-, 3^-]$, où i^+ est le sommet de cueillette et i^- est le sommet de livraison de la requête i (note : dans la représentation utilisée par l'algorithme de programmation dynamique, on introduit le dépôt 0 à la fin de chaque séquence, voir plus bas). Une multitude de tournées sont alors possibles, comme par exemple $[1^+, 1^-, 2^+, 2^-, 3^+, 4^+, 4^-, 3^-]$ ou encore $[3^+, 4^+, 4^-, 3^-, 1^+, 1^-, 2^+, 2^-]$. L'algorithme développé dans cette section se charge de trouver la tournée optimale qui respecte les séquences de chargements et déchargements dans les piles.

La notation suivante est utilisée afin de représenter le problème sous forme d'un

programme dynamique :

Soit :

- $|K| = m$: le nombre de piles
- $c(x,y)$: le coût (distance) entre les sommets x et y ;
- M_k : la séquence de chargements et déchargements associé à la pile k, $1 \leq k \leq m$, avec le dépôt 0 à la fin ;
- $M = \{M_1, ..., M_k, ..., M_m\}$: l'ensemble des séquences ;
- $T = [t_1, t_2, ..., t_k, ..., t_m]$: un vecteur de pointeurs où t_k pointe à une certaine position dans M_k, $1 \leq k \leq m$;
- M_{k,t_k} : le sommet à la position t_k dans M_k, $1 \leq k \leq m$;
- $h_k(T)$: une fonction incrémentant le pointeur associé à M_k ; T devient ainsi $[t_1, ..., t_k + 1, ..., t_m]$;

La relation de récurrence à résoudre est la suivante :

- $f^*(x,T) = \min_{k \in K, M_{k,t_k} \neq 0} \{c(x, M_{k,t_k}) + f^*(M_{k,t_k}, h_k(T))\}$ si $M_{k,t_k} \neq 0$ pour au moins un $k, 1 \leq k \leq m$,
- $f^*(x,T) = c(x,0)$ sinon.

Au départ, la tournée courante est vide, le sommet x est le dépôt 0 et T pointe sur la première position dans chaque séquence de chargements et déchargements. Une étape correspond alors au choix de l'un des sommets indiqués par les pointeurs et à son ajout à la fin de la tournée courante. Par ailleurs, l'ensemble des états à une étape donnée correspond aux configurations possibles des pointeurs. La récurrence s'interrompt lorsque chacun des pointeurs indique la dernière position de chaque séquence, où se trouve le

dépôt 0. Il faut noter que même si les séquences de chargements et déchargements sont fixées, le problème consistant à déterminer la tournée optimale correspondante demeure un problème NP-complet, dont la complexité croît selon la factorielle du nombre de sommets.

CHAPITRE 5

EXPÉRIMENTATIONS ET RÉSULTATS

Cette section présente les résultats numériques obtenus avec des instances que nous avons spécifiquement conçues pour le problème $1 - PDMS$, ainsi qu'avec des instances que l'on retrouve dans la littérature et qui correspondent à des cas particuliers de notre problème. La première partie du chapitre sera consacrée à décrire la conception de nos instances tests. Les parties suivantes traiteront de la paramétrisation de l'algorithme à grand voisinage adaptatif. Ensuite, les résultats finaux seront présentés.

5.1 Conception des instances tests

Notre ensemble d'instances tests fut conçu à partir de celles de Carrabs, Cordeau et Laporte (2007) qui contiennent un nombre de sommets (incluant le dépôt) compris entre 25 et 751. Ces instances originales furent utilisées afin de créer trois classes distinctes ayant chacune leurs caractéristiques particulières.

Le tableau 5.1 présente ces caractérisques. La première classe C1 permet d'évaluer l'avantage d'ajouter une deuxième pile à une pile existante (toutes deux de capacité infinie). La seconde classe C2 possède des demandes unitaires, considère un nombre de piles variant entre 2 et 5 et introduit une contrainte de capacité. Dans la dernière classe C3, les demandes varient entre 1 et 10, le nombre de piles varie toujours entre 2 et 5 et la contrainte de capacité est plus forte. Ici, le problème de placement des sommets dans les piles a plus d'importance que pour la classe C2. En fait, il s'agit essentiellement d'un problème de type "bin packing".

Il faut noter enfin que chaque instance dans C3 a une instance correspondante dans C2 avec le même nombre de piles.

Instance	$2n+1$	*C1*		*C2*		*C3*	
		# Piles	*Capacité*	*# Piles*	*Capacité*	*# Piles*	*Capacité*
brd14051	25	2	∞	3	2	3	16
	51	2	∞	3	3	3	14
	75	2	∞	3	4	3	18
	101	2	∞	4	2	4	10
	251	2	∞	3	14	3	28
	501	2	∞	3	20	3	35
	751	2	∞	2	25	2	35
d15112	25	2	∞	3	2	3	12
	51	2	∞	4	1	4	10
	75	2	∞	3	4	3	18
	101	2	∞	2	6	2	22
	251	2	∞	2	14	2	28
	501	2	∞	3	20	3	35
	751	2	∞	3	25	3	35
d18512	25	2	∞	3	2	3	12
	51	2	∞	5	2	5	10
	75	2	∞	2	4	2	18
	101	2	∞	4	1	4	10
	251	2	∞	3	14	3	28
	501	2	∞	2	5	2	18
	751	2	∞	3	25	3	35
fnl4461	25	2	∞	3	2	3	12
	51	2	∞	3	3	3	14
	75	2	∞	5	2	5	13
	101	2	∞	3	6	3	22
	251	2	∞	4	5	4	19
	501	2	∞	2	20	2	35
	751	2	∞	3	25	3	35
nrw1379	25	2	∞	2	1	2	10
	51	2	∞	4	2	4	10
	75	2	∞	3	8	3	10
	101	2	∞	2	11	2	10
	251	2	∞	2	6	2	10
	501	2	∞	2	10	2	10
	751	2	∞	3	15	3	10
pr1002	25	2	∞	3	1	3	10
	51	2	∞	3	2	3	14
	75	2	∞	4	2	4	10
	101	2	∞	3	4	3	20
	251	2	∞	3	4	3	14
	501	2	∞	3	8	3	25
	751	2	∞	2	8	2	25

Tableau 5.1 – Caractéristiques des instances 1 – *PDMS*

5.2 Paramètres

Les sous-sections suivantes identifient des valeurs intéressantes pour les paramètres de l'algorithme à grand voisinage adaptatif. Nous avons réalisé des tests intensifs où 16 instances furent choisies de manière à ce que les 3 classes d'instances soient représentées, en nous restreignant aux instances où le nombre de sommets se situe entre 25 et 251. Les instances avec un plus grand nombre de sommets demandent en effet un temps d'exécution considérable.

Pour chaque jeu de paramètres, l'algorithme est exécuté à 10 reprises sur chaque instance test, avec 25 000 itérations à chaque reprise, et la valeur moyenne des solutions est conservée. De plus, la meilleure valeur obtenue sur l'ensemble des exécutions est également conservée.

5.2.1 Pourcentage de destruction

Le pourcentage de destruction agit directement sur le temps d'exécution et sur la qualité des solutions. En premier lieu, il est facile de noter que le nombre de requêtes à retirer a un impact direct sur le temps d'exécution de l'algorithme. En effet, lors du retrait d'un grand nombre de requêtes, un temps important est requis pour reconstruire la solution. Cependant, il est plus difficile d'évaluer l'impact sur la qualité de la solution. Cette section a donc pour but de comparer différentes stratégies afin de trouver celle qui offre le meilleur compromis entre le temps d'exécution et la qualité de la solution.

Afin d'identifier le nombre de requêtes à retirer de la solution courante, quatre stratégies furent testées. Celles-ci utilisent toutes quatre paramètres, soit deux paramètres p_{min} et p_{max}, qui correspondent aux fractions minimales et maximales de requêtes à retirer, ainsi que deux paramètres n_{min} et n_{max} qui correspondent à des nombres absolus minimaux et maximaux de requêtes à retirer. À chaque itération de l'algorithme, le nombre de requêtes à retirer est obtenu en choisissant de manière aléatoire une valeur q satisfaisant $0 < \min\{n \cdot p_{min}, n_{min}\} \leq q \leq \min\{n \cdot p_{max}, n_{max}\}$.

La première stratégie pose $p_{min} = 1$, $n_{min} = 1$, $n_{max} = \infty$ et $p_{max} = \alpha$ où α est une constante entre 0 et 1. Puisque la borne inférieure est de 1, le nombre de requêtes à retirer

se situe dans l'intervalle $[1, n \cdot p_{max}]$. Par exemple, en posant $\alpha = 0,35$, l'algorithme retire un maximum de 35% de requêtes de la solution à chaque itération. Pour chacun des tests effectués avec cette stratégie, α varie de 0,05 à 1 avec un incrément de 0,05.

La deuxième stratégie pose $n_{min} = n_{max} = \infty$ et $p_{min} = p_{max} = \beta$. Une fraction constante de requêtes est donc retirée à chaque itération. Des tests avec β variant de 0,05 à 0,75 avec un incrément de 0,05 furent réalisés, ainsi qu'un test avec $\beta = 1$.

Dès les premiers tests, deux constatations se sont imposées. D'abord, très peu de nouvelles meilleures solutions sont atteintes lors du retrait d'un faible nombre de requêtes. Ensuite, les temps de calcul augmentent rapidement avec un trop grand nombre de requêtes. Une troisième stratégie tente donc de palier à ce problème en imposant à la fois une fraction minimale et une fraction maximale de requêtes, soit $n_{min} = n_{max} = \infty$, $p_{min} = \gamma$ et $p_{max} = \theta$. Le nombre de requêtes sera donc choisi dans l'intervalle $[n \cdot \gamma, n \cdot \theta]$. Pour les tests, différentes valeurs furent attribuées à γ et θ.

Un autre problème est observé alors pour les instances de plus grande taille. En utilisant seulement des bornes qui dépendent du nombre de requêtes n, les temps de calcul peuvent devenir prohibitifs. Une dernière stratégie borne donc également le nombre absolu de requêtes à retirer en posant $p_{min} = \gamma$, $p_{max} = \theta$, $n_{min} = \alpha$ et $n_{max} = \beta$, où $\alpha, \beta, \gamma, \theta$ sont des constantes. L'intervalle pour la sélection du nombre de requêtes à retirer devient $[\min\{\alpha, n \cdot \gamma\}, \min\{\beta, n \cdot \theta\}]$. En choisissant judicieusement les constantes, il est possible d'obtenir des temps d'exécution substantiellement réduits tout en conservant une qualité de solution semblable à la troisième stratégie.

Les résultats des deux premières stratégies sont présentés au tableau 5.2, alors que les deux dernières se retrouvent au tableau 5.3. La première colonne dans chacun de ces tableaux contient le numéro de la stratégie ; la deuxième colonne contient l'intervalle pour la sélection du nombre de requêtes à retirer ; la troisième colonne correspond à la qualité de la solution obtenue, exprimée comme la moyenne des écarts en pourcentage entre la meilleure solution obtenue après 10 exécutions sur une instance donnée et la meilleure solution connue pour cette instance sur l'ensemble des 16 instances tests ; la quatrième colonne correspond au temps d'exécution en secondes. Il faut noter que les stratégies avec les numéros 1 à 20, 21 à 36, 37 à 42 et 43 à 54 correspondent à différentes

variantes des première, deuxième, troisième et quatrième stratégies, respectivement.

Pour la première stratégie, il est facile de voir que la qualité des solutions atteint un palier quel que soit la fraction choisie entre 0,5 et 1. Évidemment, augmenter cette fraction a seulement pour effet d'accroître les temps de calcul. Avec 0,5, par exemple, il est possible d'obtenir une qualité similaire à 1 mais avec un temps d'exécution deux fois plus rapide.

Les résultats de la deuxième stratégie sont sensiblement meilleurs que ceux de la première stratégie pour des fractions de requêtes variant de 0,05 à 0,55, mais les temps de calcul sont plus importants. Par ailleurs, les meilleurs résultats sont obtenus pour des valeurs variant entre 0,30 et 0,55. Ces résultats indiquent que le retrait d'un nombre trop petit ou trop grand de requêtes ne permet pas d'atteindre les meilleures solutions possibles. En effet, un nombre trop petit ne permet pas d'explorer le voisinage de façon adéquate, alors qu'un nombre trop grand transforme l'algorithme en une simple méthode de construction qui mène à des solutions de qualité médiocre. Nous avons également observé que cette stratégie est très lente sur les plus grandes instances. On peut voir au Tableau 5.3 que la troisième stratégie arrive à produire de meilleurs résultats que les deux premières stratégies. Toutefois, cette stratégie laisse également à désirer sur les plus grandes instances. La quatrième stratégie pallie à cette difficulté en obtenant une qualité de solution qui se compare avantageusement à la troisième stratégie, mais avec des temps de calcul réduits. Il est cependant difficile de choisir un intervalle de valeurs particulier car la qualité des solutions varie peu, soit entre 1,62% et 1,94%. Bien sûr, plus l'intervalle autorise le retrait d'un grand nombre de requêtes, plus le temps d'exécution augmente. En comparant les stratégies numéros 43 et 54, on voit que le gain additionnel obtenu avec 54 est de 0,08% seulement, mais plus du double du temps d'exécution est requis. Ainsi, la quatrième stratégie avec l'intervalle $[min\{10;0,15n\};min\{35;0,45n\}]$ semble offrir un compromis intéressant entre la qualité de la solution et le temps d'exécution et elle a été retenue pour le reste des tests.

No	*Intervalle*	*Solution*	*Temps (s)*		*No*	*Intervalle*	*Solution*	*Temps (s)*
1	$[1;0,05n]$	19,23%	1,1		21	$0,05n$	13,81%	1,9
2	$[1;0,10n]$	8,75%	2,2		22	$0,10n$	4,57%	5,4
3	$[1;0,15n]$	4,76%	3,9		23	$0,15n$	3,42%	10,4
4	$[1;0,20n]$	3,62%	6,8		24	$0,20n$	2,75%	19,7
5	$[1;0,25n]$	3,06%	9,6		25	$0,25n$	2,06%	28,7
6	$[1;0,30n]$	2,77%	13,3		26	$0,30n$	1,82%	39,3
7	$[1;0,35n]$	2,55%	17,4		27	$0,35n$	1,80%	51,4
8	$[1;0,40n]$	2,19%	22,0		28	$0,40n$	1,78%	64,4
9	$[1;0,45n]$	2,11%	26,6		29	$0,45n$	1,73%	75,0
10	$[1;0,50n]$	1,99%	31,9		30	$0,50n$	1,88%	87,0
11	$[1;0,55n]$	1,96%	37,1		31	$0,55n$	1,88%	96,4
12	$[1;0,60n]$	1,88%	42,4		32	$0,60n$	2,12%	105,3
13	$[1;0,65n]$	2,03%	47,2		33	$0,65n$	2,35%	113,1
14	$[1;0,70n]$	1,75%	52,9		34	$0,70n$	2,85%	121,0
15	$[1;0,75n]$	2,04%	57,8		35	$0,75n$	3,29%	126,7
16	$[1;0,80n]$	1,83%	62,3		36	n	10,84%	128,5
17	$[1;0,85n]$	1,73%	67,6					
18	$[1;0,90n]$	1,88%	71,1					
19	$[1;0,95n]$	1,94%	74,2					
20	$[1;n]$	1,89%	77,3					

Tableau 5.2 – Qualité des solutions et temps d'exécution par rapport au nombre de requêtes retirées pour les stratégies 1 et 2

No	*Intervalle*	*Solution*	*Temps (s)*
37	$[0,10n;0,40n]$	1,96%	28,9
38	$[0,15n;0,45n]$	1,74%	38,2
39	$[0,20n;0,50n]$	1,82%	51,0
40	$[0,25n;0,55n]$	1,79%	62,3
41	$[0,30n;0,55n]$	1,69%	68,1
42	$[0,30n;0,60n]$	1,67%	73,9
43	$[min\{10;0,15n\};min\{35;0,45n\}]$	1,85%	30,6
44	$[min\{15;0,15n\};min\{40;0,45n\}]$	1,72%	33,0
45	$[min\{20;0,15n\};min\{45;0,45n\}]$	1,93%	35,0
46	$[min\{30;0,15n\};min\{50;0,45n\}]$	1,86%	36,1
47	$[min\{10;0,20n\};min\{35;0,55n\}]$	1,76%	41,2
48	$[min\{15;0,20n\};min\{40;0,55n\}]$	1,80%	46,0
49	$[min\{20;0,20n\};min\{45;0,55n\}]$	1,94%	48,9
50	$[min\{30;0,20n\};min\{50;0,55n\}]$	1,62%	51,1
51	$[min\{10;0,30n\};min\{35;0,60n\}]$	1,82%	43,9
52	$[min\{15;0,30n\};min\{40;0,60n\}]$	1,69%	55,0
53	$[min\{20;0,30n\};min\{45;0,60n\}]$	1,75%	60,0
54	$[min\{30;0,30n\};min\{50;0,60n\}]$	1,77%	64,0

Tableau 5.3 – Qualité des solutions et temps d'exécution par rapport au nombre de requêtes retirées pour les stratégies 3 et 4

5.2.2 Sélection des opérateurs

Pour cette étape du paramétrage, le but est d'identifier de bonnes valeurs pour les paramètres σ_1, σ_2 et σ_3. L'algorithme *ALNS* utilise ces paramètres pour mettre à jour le poids des opérateurs lors de succès ou d'échecs. Dans l'algorithme, la valeur σ_1 sert à bonifier les opérateurs menant à de nouvelles meilleures solutions. Le second paramètre σ_2 bonifie les nouvelles solutions acceptées qui sont meilleures que la solution courante. Le dernier paramètre σ_3 augmente le poids de l'opérateur lorsqu'une solution de moins bonne qualité est acceptée.

Il est facile de penser qu'une paramétrisation précise de ces valeurs permettra de sélectionner les opérateurs ayant le plus grand impact sur la recherche. Par exemple, une augmentation des poids des opérateurs lors de l'acceptation d'une moins bonne solution produit une forme de diversification et évite de rester coinçé dans des minima locaux. Par ailleurs, une augmentation des poids des opérateurs par σ_1 et σ_2 intensifie la recherche.

Un total de 9 jeux de valeurs furent testés pour σ_1, σ_2 et σ_3. Le tableau 5.4 présente ces valeurs, la qualité des solutions obtenues et le temps d'exécution en secondes. Le premier jeu consiste à poser $\sigma_1 = \sigma_2 = \sigma_3 = 0$ de façon à obtenir une sélection aléatoire des opérateurs. On peut ainsi savoir s'il y a un avantage à affecter des poids aux opérateurs de manière adaptative. Pour les 5 jeux suivants, des valeurs assez naturelles furent utilisées où $\sigma_1 > \sigma_2 > \sigma_3$. Les jeux 7 et 8 utilisent l'argument de Pisinger et Ropke (2007) selon lequel il faut bonifier les opérateurs permettant de visiter de nouvelles solutions de moindre qualité afin de diversifier la recherche. Le huitième jeu correspond aux valeurs proposées par Pisinger et Ropke (2007). Le dernier jeu propose des valeurs allant à l'encontre des hypothèses de départ en posant $\sigma_1 < \sigma_3$.

Les résultats présentés dans le tableau montrent que ces paramètres n'ont pas un impact majeur sur la qualité des solutions. On note toutefois que les jeux où la constante σ_3 est posée à zéro permettent de produire des solutions qui sont généralement de meilleure qualité. De plus, les jeux avec un grand σ_1, comme les jeux 3, 4 et 5, requièrent un temps d'exécution un peu plus long que les jeux 1 et 2 car l'opérateur d'insertion avec regrets, qui est assez coûteux en temps de calcul, produit souvent de nouvelles meilleures solu-

tions. Cette situation étant rétribuée par la valeur σ_1, cet opérateur aura donc tendance à être plus fortement favorisé lors du processus de sélection.

Des analyses plus approfondies furent effectués avec les jeux 1, 2, 3, 4 et 5, en considérant les plus grandes instances tests disponibles (avec 751 sommets). Il est alors apparu que le jeu $\sigma_1 = 4$, $\sigma_2 = 1$ et $\sigma_3 = 0$ offrait un compromis intéressant entre temps d'exécution et qualité de solution (bien que la qualité des solutions était encore une fois assez peu sensible au jeu considéré). Au final, c'est donc le jeu $\sigma_1 = 4$, $\sigma_2 = 1$ et $\sigma_3 = 0$ qui fut choisi.

5.2.3 Contribution de l'algorithme de programmation dynamique

À la fin de la métaheuristique à grand voisinage adaptatif, l'algorithme de programmation dynamique est lancé. Cet algorithme produit la tournée optimale avec le placement des sommets dans les piles associé à la meilleure solution rencontrée. Les tests démontrent que cet algorithme améliore la solution de 0,16% en moyenne pour un temps d'exécution moyen de seulement 0,04 seconde.

5.2.4 Contribution des opérateurs

Cette section analyse la contribution de chacun des opérateurs sur la qualité des solutions obtenues. Chacun des tests utilise un sous-ensemble d'opérateurs pour l'insertion et pour le retrait. Les résultats sont présentés au tableau 5.5.

No	σ_1	σ_2	σ_3	Qualité	Temps (s)
1	0	0	0	1,52%	23,4
2	1	0	0	1,67%	23,5
3	4	1	0	1,57%	29,3
4	5	1	0	1,55%	29,8
5	7	2	0	1,47%	29,6
6	9	4	1	1,67%	29,1
7	9	2	7	1,65%	25,7
8	33	9	13	1,82%	26,8
9	1	0	10	1,90%	22,1

Tableau 5.4 – Qualité des solutions en fonction des paramètres σ_1, σ_2 et σ_3

La première partie consiste à utiliser seulement l'opérateur d'insertion à moindre coût avec un seul opérateur de retrait. Les résultats montrent que l'opérateur de retrait orienté vers les piles est le meilleur des quatre. Toutefois, son temps d'exécution est légèrement plus élevé. Ceci s'explique par la plus grande complexité de l'opérateur. Un test comprenant les 4 opérateurs de retrait fut également tenté pour voir si leur combinaison apporte une amélioration. Malheureusement, ce n'est pas le cas, et il est donc plus simple de prendre seulement l'opérateur de retrait orienté vers les piles.

Une deuxième expérimentation de la même sorte exploitant seulement l'opérateur d'insertion avec regrets fut réalisée. L'apport des opérateurs de retrait est le même que pour la première expérimentation. Ainsi, dans un contexte d'algorithme à grand voisinage (avec seulement un opérateur d'insertion et un opérateur de retrait), il est préférable de choisir l'opérateur de retrait orienté vers les piles. Un gain d'au moins 0,2% par rapport aux autres opérateurs de retrait est alors envisageable. Toutefois, l'utilisation combinée des quatre opérateurs de retrait permet d'obtenir un gain additionnel de 0,1% par rapport à l'utilisation seule de l'opérateur sur les piles.

Deux tests furent effectués en éliminant l'opérateur de retrait orienté vers la tournée car les résultats montrent que celui-ci est le pire des quatre opérateurs de retrait. Contrairement à ce qui était attendu, son utilisation en combinaison avec les autres apporte un gain d'au moins 0,1%. Il semble donc qu'il ait son utilité dans l'exploration de l'espace des solutions.

La dernière expérimentation étudie l'apport des opérateurs d'insertion. Les résultats démontrent que l'opérateur d'insertion à moindre coût combiné avec celui basé sur les regrets ne permet pas d'améliorer les résultats. Toutefois, les temps de calculs sont réduits d'environ 21%, pour une diminution de qualité de 0,03% seulement, lorsque les quatre opérateurs de retrait sont utilisés. Dans le cas où l'opérateur de retrait basé sur les piles est le seul à être utilisé, la qualité diminue toutefois de 0,2%, ce qui est déjà plus considérable.

Au final, tous les opérateurs d'insertion et de retrait furent retenus. En effet, même si l'utilisation seule de l'opérateur d'insertion avec regrets apporte une légère amélioration à la qualité des solutions, les temps d'exécution augmentent de 25% par rapport à

l'utilisation des deux opérateurs d'insertion.

5.3 Présentation des résultats finaux

Les sous-sections suivantes présentent les résultats finaux sur toutes les instances $1-PDMS$, $DTSPMS$ et $TSPPDL$. Il faut noter que tous les tests furent effectués sur un ordinateur avec un processeur AMD Opteron 275 cadencé à 2.2 Ghz. L'algorithme fut codé en C et compilé à l'aide de gcc.

L'accronyme *ALNS* fut utilisé afin d'identifier l'algorithme à grand voisinage adaptatif. Pour tous les types de problèmes, chaque exécution de l'algorithme comprend 25 000 itérations. De plus, une version réduite de cet algorithme nommé *LNS* fut créée afin de comparer les résultats. Celle-ci utilise seulement l'opérateur d'insertion à moindre coût et l'opérateur de retrait aléatoire ce qui lui permet d'être très simple et rapide en temps d'exécution. En fait, cet algorithme est fort possiblement l'implantation la plus simple de la métaheuristique de Pisinger et Ropke (2007). Les valeurs des paramètres utilisées pour *LNS* sont exactement les mêmes que pour les paramètres correspondants dans *ALNS*.

5.3.1 Résultats $1-PDMS$

Cette sous-section rapporte les résultats obtenus sur les instances que nous avons conçues pour ce travail. Celles-ci se divisent en trois classes. La classe C1 se contente d'ajouter une pile aux instances originales de Carrabs, Cordeau et Laporte (2007), et il n'y a pas de contrainte de capacité. La classe C2 contient des instances avec des requêtes de demande unitaire et où le nombre de piles varie entre 2 et 5. De plus une contrainte de capacité est présente. Pour la classe C3, les demandes varient entre 1 et 10, le nombre de piles varie toujours entre 2 et 5 et la contrainte de capacité est beaucoup plus liante.

Les deux algorithmes *LNS* et *ALNS* furent exécutés 10 fois sur chaque instance. L'écart en pourcentage de la meilleure des 10 exécutions par rapport à la meilleure solution connue ("Meilleure"), l'écart moyen ("Moy.") et le temps d'exécution moyen en secondes (pour une exécution) sont rapportés dans les tableaux qui suivent pour cha-

Opérateur d'insertion	Opérateur de retrait	Qualité	Temps (s)
Moindre coût	Aléatoire	2,843%	5,3
	Distance	2,721%	9,0
	Route	4,807%	5,1
	Piles	2,640%	8,9
	Tous	2,702%	6,9
	Tous moins Route	2,805%	7,1
Regrets	Aléatoire	1,868%	51,5
	Distance	1,919%	54,4
	Route	3,683%	49,6
	Piles	1,639%	55,1
	Tous	1,629%	53,0
	Tous moins Route	1,884%	53,2
Tous	Tous moins Route	1,739%	43,9
	Piles	1,844%	46,0
	Tous	1,654%	43,7

Tableau 5.5 – Qualité des solutions en fonction des opérateurs

cun des deux algorithmes. Les meilleures solutions connues furent obtenues en prenant la meilleure de 10 exécutions de l'algorithme *ALNS* avec 50 000 itérations à chaque exécution.

Les tableaux 5.6, 5.7 et 5.8 rapportent les résultats sur les instances $1-PDMS$. La première colonne indique le nom de l'instance, la deuxième contient le nombre de sommets, la troisième et la quatrième affichent le nombre de piles et la capacité. La colonne suivante indique les valeurs des meilleures solutions connues. Les six dernières colonnes comparent les deux algorithmes à la meilleure solution connue.

Les résultats pour la classe C1 se retrouvent au tableau 5.6. En comparant *LNS* à *ALNS*, on voit que *LNS* est en moyenne 5 fois plus rapide. La différence en terme de qualité moyenne des solutions est de 0,4% ce qui reste faible pour un algorithme 5 fois plus rapide. Toutefois, la différence s'accentue à mesure que la taille des instances augmente. Ainsi, *LNS* s'exécute jusqu'à 9 fois plus rapidement pour une différence de qualité de plus de 1% sur les plus grandes instances. Globalement, l'algorithme *ALNS* se situe en moyenne à 1,41% de la meilleure solution connue pour un temps d'exécution moyen de 251 secondes. Pour ce qui est de *LNS*, l'algorithme requiert un temps moyen de 34 secondes pour un écart moyen de 1,83%.

Les résultats de la classe C2 se trouvent dans le tableau 5.7. On observe d'abord

que les résultats pour cette classe sont beaucoup plus éloignés des meilleures solutions connues que ceux de la classe C1. Ainsi, *ALNS* produit un écart moyen de 3,40% alors que *LNS* se situe à 7%. Avec ces résultats, il est possible de dire que *LNS* est particulièrement faible pour les instances dont les piles sont de capacité finie. Sur certaines instances, la moyenne se situe à plus de 20% de la meilleure solution connue. L'algorithme *LNS* possède quand même l'avantage d'être en moyenne 9 fois plus rapide que *ALNS* sur ces instances.

Le tableau 5.8 rapporte les résultats pour la classe C3. De manière globale, l'algorithme *LNS* obtient une solution moyenne qui est à 9,73% de la meilleure solution connue en un temps moyen de 35 secondes alors que *ALNS* est à 3,49% en un temps moyen de 309 secondes. Les résultats pour la classe C3 sont un peu moins bons que ceux pour la classe C2, en particulier pour l'algorithme *LNS*. Ceci indique que les instances de la classe C3 sont plus difficiles à résoudre. Pour les instances de grande taille, la qualité des solutions produites par *LNS* est 11% moins bonne que *ALNS*, ce qui permet d'affirmer que *LNS* est mal adapté aux instances ayant une contrainte de capacité forte.

D'après les tests effectués, l'algorithme *ALNS* se comporte de manière plus stable sur les instances $1-PDMS$ que *LNS*. Ceci s'explique sans doute par l'utilisation de l'opérateur d'insertion avec regrets qui est fort utile pour les problèmes à plusieurs piles. Toutefois, *LNS* possède l'avantage d'être d'une grande rapidité : il est possible d'obtenir une solution en 2 à 3 minutes de calcul pour les plus grandes instances à 751 sommets, comparativement à 45 minutes pour *ALNS*.

Un autre aspect intéressant à considérer est l'avantage d'utiliser deux piles au lieu d'une seule. Le tableau 5.9 compare les meilleures solutions obtenues sur les instances à une pile de Carrabs, Cordeau et Laporte (2007) et celles de la classe C1 où une deuxième pile a été ajoutée. Les résultats montrent que l'ajout d'une pile diminue en moyenne de 22% la distance totale. Pour les problèmes de taille supérieure à 251 sommets, cette moyenne se situe autour de 28%. Ainsi, l'ajout d'une deuxième pile apporte une amélioration importante.

Instance	$2n+1$	*#Piles*	*Cap.*	*Meilleure connue*	*LNS*			*ALNS*		
					Meilleure	*Moy.*	*Temps (s)*	*Meilleure*	*Moy.*	*Temps (s)*
brd14051	25	2	∞	4299	0,00	0,00	0,3	0,00	0,00	0,4
	51	2	∞	6659	0,00	0,21	1,1	0,00	0,01	2,9
	75	2	∞	5772	0,24	0,87	2,4	0,00	0,57	9,4
	101	2	∞	7985	0,24	0,58	4,6	0,04	0,19	25,1
	251	2	∞	16138	0,62	2,08	20,9	0,10	1,11	184,4
	501	2	∞	35920	0,81	2,99	69,3	0,00	2,43	519,2
	751	2	∞	59446	1,44	2,88	148,4	0,85	2,77	1112,5
d15112	25	2	∞	78806	0,00	0,00	0,3	0,00	0,00	0,4
	51	2	∞	120670	0,00	0,16	1,1	0,00	0,09	2,9
	75	2	∞	153815	0,00	0,45	2,5	0,00	0,50	10,3
	101	2	∞	202967	0,42	1,33	4,6	0,00	0,46	26,8
	251	2	∞	409313	0,92	2,61	20,8	0,00	1,33	182,4
	501	2	∞	659044	1,37	3,81	69,0	3,00	4,15	548,0
	751	2	∞	942681	1,68	3,44	132,9	2,75	3,56	1126,9
d18512	25	2	∞	4299	0,00	0,00	0,3	0,00	0,00	0,4
	51	2	∞	6638	0,21	0,26	1,1	0,00	0,19	2,8
	75	2	∞	7315	0,00	1,40	2,3	0,00	0,93	9,0
	101	2	∞	8256	0,29	1,56	4,5	0,13	0,74	25,3
	251	2	∞	16862	1,83	5,09	20,6	0,44	3,53	178,0
	501	2	∞	35167	2,32	3,92	66,3	0,51	2,47	517,0
	751	2	∞	58078	1,66	3,85	126,0	1,43	3,38	938,4
fnl4461	25	2	∞	1953	0,00	0,00	0,3	0,00	0,00	0,4
	51	2	∞	3225	0,00	0,00	1,1	0,00	0,00	3,1
	75	2	∞	4457	0,00	0,27	2,5	0,00	0,02	10,2
	101	2	∞	6243	0,00	1,96	4,4	1,39	2,10	24,9
	251	2	∞	21944	0,55	2,87	20,7	0,00	1,60	180,9
	501	2	∞	49311	2,48	4,27	69,7	2,38	3,70	489,8
	751	2	∞	80999	3,08	4,88	140,7	1,75	3,21	918,6
nrw1379	25	2	∞	2894	0,00	0,00	0,3	0,00	0,00	0,4
	51	2	∞	4099	0,00	0,00	1,1	0,00	0,00	3,2
	75	2	∞	5217	0,08	0,26	2,4	0,17	0,37	9,8
	101	2	∞	6924	0,00	0,31	4,4	0,33	0,33	24,4
	251	2	∞	18900	2,83	4,94	20,8	0,00	2,93	176,0
	501	2	∞	41989	2,59	4,26	76,5	2,19	4,02	552,4
	751	2	∞	75034	1,45	3,00	145,4	0,85	1,84	1035,8
pr1002	25	2	∞	14966	0,00	0,00	0,3	0,00	0,00	0,4
	51	2	∞	25112	0,21	0,23	1,1	0,00	0,22	3,1
	75	2	∞	36327	0,00	0,60	2,3	0,00	0,28	9,4
	101	2	∞	47336	0,00	0,37	4,5	0,00	0,82	25,2
	251	2	∞	140231	0,94	2,98	20,6	1,21	2,59	178,6
	501	2	∞	335203	1,99	3,56	74,8	0,69	3,22	501,5
	751	2	∞	566059	3,95	4,78	144,8	1,56	3,55	985,9
Moyenne					0,81	1,83	34,2	0,52	1,41	251,3
Moy. ≥ 251					1,81	3,68	77,1	1,09	2,85	573,7

Tableau 5.6 – Résultats sur les instances 1 – *PDMS* de la classe C1

Instance	$2n+1$	*#Piles*	*Cap.*	*Meilleure connue*	*LNS*			*ALNS*		
					Meilleure	*Moy.*	*Temps (s)*	*Meilleure*	*Moy.*	*Temps (s)*
brd14051	25	3	2	4293	0,07	0,67	0,2	0,00	0,19	0,3
	51	3	3	6754	0,68	1,30	1,0	0,38	0,68	3,1
	75	3	4	7030	0,83	2,11	2,9	0,01	0,19	13,6
	101	4	2	8640	13,03	21,21	2,6	4,14	5,65	18,1
	251	3	14	14068	3,62	5,59	33,4	1,36	2,17	353,6
	501	3	20	29840	5,06	7,65	101,6	0,79	4,05	953,8
	751	2	25	60321	4,55	5,36	134,5	1,33	2,76	1127,6
d15112	25	3	2	77130	1,84	2,44	0,2	1,53	2,64	0,3
	51	4	1	138313	6,08	8,55	0,7	0,00	2,64	2,3
	75	3	4	142609	0,33	1,27	3,1	0,29	0,81	14,0
	101	2	6	205404	1,08	2,23	4,1	0,28	1,30	23,8
	251	2	14	407199	0,41	3,91	20,2	0,00	2,33	186,4
	501	3	20	563446	2,22	5,41	98,0	1,44	3,05	959,0
	751	3	25	793770	3,54	4,96	186,4	1,39	3,84	1785,7
d18512	25	3	2	4293	0,07	0,67	0,2	0,00	0,19	0,3
	51	5	2	6280	3,55	6,69	1,0	0,97	1,71	3,3
	75	2	4	7535	2,10	3,05	1,8	0,78	1,26	6,9
	101	4	1	13954	5,17	8,85	2,3	0,44	4,17	15,1
	251	3	14	14641	7,79	9,90	32,8	0,05	2,17	343,3
	501	2	5	49178	19,84	21,26	53,2	9,92	12,34	398,3
	751	3	25	48343	5,49	7,47	176,5	3,87	5,02	1903,2
fnl4461	25	3	2	2008	0,05	0,27	0,2	0,05	0,86	0,4
	51	3	3	3042	0,36	1,54	1,1	0,00	0,86	3,0
	75	5	2	4129	15,86	19,83	2,1	1,40	8,38	11,2
	101	3	6	5475	1,86	3,96	6,3	0,97	1,39	40,5
	251	4	5	18950	13,29	14,09	35,8	2,24	5,75	390,1
	501	2	20	50148	6,95	8,36	64,9	1,53	3,53	512,4
	751	3	25	68542	5,28	6,32	175,2	2,54	3,99	1707,6
nrw1379	25	2	1	3548	0,06	0,28	0,2	0,00	0,08	0,2
	51	4	2	4003	7,49	12,63	0,8	0,62	3,57	2,8
	75	3	8	4680	0,28	0,41	3,7	0,11	0,25	17,1
	101	2	11	6982	0,03	1,98	4,2	0,00	1,28	24,1
	251	2	6	21212	10,82	12,60	16,9	3,05	4,76	156,7
	501	2	10	43993	5,37	7,86	68,4	5,77	6,32	499,0
	751	3	15	62645	6,41	7,95	199,9	3,91	6,29	1683,1
pr1002	25	3	1	20078	0,00	0,17	0,2	0,00	0,00	0,3
	51	3	2	27023	4,78	7,11	0,7	2,88	4,63	2,1
	75	4	2	36921	8,24	13,19	1,5	1,94	6,46	7,2
	101	3	4	45254	2,76	5,65	5,2	0,00	2,73	32,3
	251	3	4	145028	15,94	18,60	21,4	6,25	7,90	215,7
	501	3	8	298068	10,75	11,61	103,1	2,35	6,57	775,2
	751	2	8	635334	7,77	10,21	143,4	5,92	7,80	928,0
Moyenne					5,04	7,03	40,8	1,68	3,40	360,0
Moy. ≥ 251					7,51	9,40	92,5	2,98	5,04	826,6

Tableau 5.7 – Résultats sur les instances 1 – *PDMS* de la classe C2

Instance	$2n+1$	*#Piles*	*Cap.*	*Meilleure connue*	*LNS*			*ALNS*		
					Meilleure	*Moy.*	*Temps (s)*	*Meilleure*	*Moy.*	*Temps (s)*
brd14051	25	3	16	4293	0,02	0,32	0,3	0,00	0,20	0,4
	51	3	14	7011	0,20	1,23	1,1	0,03	0,13	3,5
	75	3	18	7077	1,23	2,81	2,9	0,27	0,69	13,7
	101	4	10	10307	4,75	7,15	4,0	0,74	2,12	27,0
	251	3	28	16810	10,92	17,96	28,6	1,79	3,28	298,7
	501	3	35	34230	15,48	16,52	93,1	3,27	5,75	831,2
	751	2	35	70292	11,23	13,42	126,2	5,33	6,75	945,7
d15112	25	3	12	85231	0,04	0,53	0,2	0,04	0,05	0,3
	51	4	10	125772	2,54	3,76	1,0	1,02	1,77	3,2
	75	3	18	146039	0,54	0,99	2,7	0,05	0,41	11,8
	101	2	22	214387	1,69	4,58	3,8	0,07	1,09	21,7
	251	2	28	444941	8,66	10,84	17,6	0,35	2,65	168,1
	501	3	35	603569	9,31	10,99	94,6	3,15	4,19	861,4
	751	3	35	864123	8,84	11,59	167,3	4,34	6,55	1644,5
d18512	25	3	12	4306	0,51	0,87	0,3	0,00	0,34	0,4
	51	5	10	6720	0,54	1,55	1,3	0,03	0,31	4,4
	75	2	18	8013	1,42	3,62	1,8	0,34	0,54	7,3
	101	4	10	11565	6,74	7,92	4,0	0,78	2,16	27,5
	251	3	28	17324	14,93	18,16	27,5	2,63	4,18	287,5
	501	2	18	62630	23,60	25,49	58,4	6,62	8,52	458,9
	751	3	35	56223	10,39	13,55	182,5	4,95	6,21	1655,8
fnl4461	25	3	12	2043	0,00	0,37	0,3	0,00	0,51	0,5
	51	3	14	3331	0,06	1,41	1,1	0,00	0,85	3,2
	75	5	13	3838	3,57	5,53	4,3	0,03	0,63	20,6
	101	3	22	6026	4,61	6,75	5,7	0,00	2,62	38,1
	251	4	19	20263	13,69	15,78	33,5	2,48	3,94	375,7
	501	2	35	60761	15,13	16,76	71,3	4,31	5,59	511,2
	751	3	35	82076	14,97	17,13	178,3	5,85	8,02	1674,4
nrw1379	25	2	10	3208	0,00	0,00	0,2	0,00	0,01	0,3
	51	4	10	3940	3,45	5,35	1,1	0,00	3,10	3,3
	75	3	10	6168	5,20	8,51	1,9	0,79	3,89	8,2
	101	2	10	12760	10,28	13,12	2,2	1,90	4,64	12,6
	251	2	10	34212	20,12	22,22	10,2	6,01	7,04	105,1
	501	2	10	78321	19,51	20,82	33,6	4,92	6,39	273,0
	751	3	10	105512	24,46	25,33	86,1	9,21	10,70	830,6
pr1002	25	3	10	20020	0,00	0,08	0,2	0,00	0,02	0,3
	51	3	14	26268	0,64	0,83	1,1	0,31	0,67	3,4
	75	4	10	37457	8,33	11,22	2,3	1,70	6,44	10,8
	101	3	20	45315	4,12	5,47	5,7	0,70	1,93	36,0
	251	3	14	181739	23,34	25,32	19,5	3,90	6,27	218,1
	501	3	25	319393	15,46	17,49	80,7	4,57	6,90	743,6
	751	2	25	704578	13,88	15,29	112,8	6,44	8,60	874,1
Moyenne					7,96	9,73	35,0	2,12	3,49	309,9
Moy. ≥ 251					15,22	17,48	79,0	4,45	6,20	708,8

Tableau 5.8 – Résultats sur les instances 1 – *PDMS* de la classe C3

Instance	$2n+1$	*Meilleure 1 Pile*	*Meilleure 2 Piles*	*Différence*
brd14051	25	4672	4299	-0,08
	51	7740	6659	-0,14
	75	7232	5772	-0,20
	101	9731	7985	-0,18
	251	22243	16138	-0,27
	501	50027	35920	-0,28
	751	82411	59446	-0,28
d15112	25	93981	78806	-0,16
	51	142113	120670	-0,15
	75	199001	153815	-0,23
	101	265191	202967	-0,23
	251	562072	409313	-0,27
	501	925880	659044	-0,29
	751	1323452	942681	-0,29
d18512	25	4672	4299	-0,08
	51	7502	6638	-0,12
	75	8629	7315	-0,15
	101	10242	8256	-0,19
	251	23243	16862	-0,27
	501	49499	35167	-0,29
	751	80869	58078	-0,28
fnl4461	25	2168	1953	-0,10
	51	4020	3225	-0,20
	75	5739	4457	-0,22
	101	8530	6243	-0,27
	251	28613	21944	-0,23
	501	68502	49311	-0,28
	751	114345	80999	-0,29
nrw1379	25	3192	2894	-0,09
	51	5055	4099	-0,19
	75	6831	5217	-0,24
	101	9817	6924	-0,29
	251	26470	18900	-0,29
	501	58478	41989	-0,28
	751	102847	75034	-0,27
pr1002	25	16221	14966	-0,08
	51	30936	25112	-0,19
	75	46600	36327	-0,22
	101	61495	47336	-0,23
	251	188960	140231	-0,26
	501	468323	335203	-0,28
	751	796861	566059	-0,29
Moyenne				-0,22
Moy. ≥ 251				-0,28

Tableau 5.9 – Comparaison entre les meilleurs solutions connues pour les problèmes à une pile et à deux piles

5.3.2 Résultats *DTSPMS*

La seconde plage de problèmes provient d'une variante proposée par Petersen et Madsen (2007) nommée "Double Traveling Salesman Problem with Multiple Stacks" ou (*DTSPMS*). Dans ce travail, la problématique consiste à concevoir une tournée de cueillettes démarrant d'un dépôt d_1 afin de remplir un conteneur. Une fois plein, ce conteneur est rapporté au dépôt d_1. Celui-ci est ensuite transporté à un autre dépôt d_2 à partir duquel sont effectuées les livraisons. Dans les instances conçues par les auteurs, le conteneur est composé de 3 piles de capacité équivalant à 1/3 du nombre de requêtes. Des instances avec 12, 33 et 66 requêtes furent proposées totalisant 60 instances tests. Les valeurs optimales sont rapportées pour les 20 instances à 12 requêtes. Pour les autres instances, les meilleures valeurs obtenues sont rapportées. Pour ces problèmes, les distances sont arrondies à l'entier le plus près.

Dans leur publication, Petersen et Madsen (2007) proposent quatre métaheuristiques afin de résoudre le problème dont trois utilisent deux structures de voisinage distinctes. Le premier voisinage consiste à modifier une tournée en échangeant deux sommets consécutifs. Si les sommets sont dans la même pile, il faut en plus échanger leur position dans la pile (ainsi que leur sommet de cueillette/livraison correspondant). Le second voisinage échange deux requêtes choisies dans deux piles différentes. Des métaheuristiques par recuit simulé, recherche tabou et recherche locale itérée furent utilisées par les auteurs. Cependant, les résultats obtenus avec ces trois algorithmes se sont révélés d'assez piètre qualité. Seuls les résultats de la dernière métaheuristique sont retenus ici car celle-ci performe largement mieux que les autres. Il s'agit d'une métaheuristique à grand voisinage faisant appel à des opérateurs d'insertion différents de ceux utilisés dans ce mémoire. Ainsi, les auteurs utilisent l'opérateur de retrait de Shaw (1998) et plusieurs opérateurs d'insertion glouton. En fait, ils font appel à quatre opérateurs : insertion au plus proche, au plus loin, à moindre coût et à pire coût. La requête choisie à chaque itération optimise un des critères précédents. De plus, à la fin de chaque itération, une descente locale est effectuée en utilisant les deux voisinages précédents.

Afin de transformer ce problème en un problème $1-PDMS$, il suffit de modifier la

matrice des distances en posant : $c_{i,j} = c_{i,d_1} + c_{d_2,j}$, $c_{j,i} = c_{d_1,j} = c_{i,d_2} = \infty \; \forall i \in P, \forall j \in D$ où d_1 est le dépôt de départ et d_2 est le dépôt d'arrivée du véhicule. La première modification pose la distance entre une cueillette et une livraison comme la distance entre la cueillette et d_1 plus la distance de d_2 au point de livraison. Il est aussi primordial de poser à l'infini la distance entre un point de livraison et de cueillette afin d'éviter une solution n'effectuant pas toutes les cueuillettes avant les livraisons. Par ailleurs, la distance entre les dépôts d_1 et d_2 est ignorée car celle-ci peut être considérée comme une constante.

Les tests de Petersen et Madsen (2007) sur ces instances furent effectués sur un Dell D610 avec un processeur cadencé à 1,6 Ghz et les algorithmes furent codés en Java 1.5. Il faut noter que la machine que nous avons utilisée dans ce travail est environ 3 fois plus rapide.

L'algorithme *ALNS* fut exécuté 10 fois sur chaque instance avec 25 000 itérations par exécution. De ces 10 exécutions, la meilleure solution fut conservée ainsi que la moyenne des solutions. Le temps de calcul moyen en secondes (pour une exécution) fut aussi conservé.

Les tableaux 5.10, 5.11, 5.12 présentent les résultats. La première colonne indique le nom de l'instance, la deuxième colonne contient la valeur de la meilleure solution obtenue par Petersen et Madsen (2007), les troisième et quatrième colonnes indiquent l'écart en pourcentage par rapport à la meilleure solution connue pour les versions courte (10 secondes) et longue (3 minutes) de l'algorithme de Petersen et Madsen (2007). Les trois colonnes suivantes contiennent les résultats de l'algorithme *LNS*, soit la meilleure solution obtenue après 10 exécutions, la solution moyenne et le temps d'exécution en secondes pour une exécution. Les trois dernières colonnes présentent enfin les résultats pour *ALNS* avec le même format que les trois colonnes précédentes.

Dans le tableau 5.10 les résultats pour les instances à 12 requêtes sont présentés. On peut noter que les trois algorithmes performent de façon similaire sur ce premier jeu de tests, soit un écart moyen de 0,21% pour *LNS* et 0,08% pour *ALNS*. Toutefois, les deux effectuent le travail en 0,3 et 0,5 seconde approximativement, ce qui est au moins 20 fois plus rapide que la version courte de l'algorithme de Petersen et Madsen qui s'exécute

en 10 secondes. De plus, la meilleure solution est optimale pour chacune des instances. Pour fins de comparaison, la moyenne obtenue par *ALNS* est sensiblement meilleure que celle de *LNS*.

Les résultats pour les instances à 33 requêtes se trouvent au tableau 5.11. La solution moyenne produite par *LNS* se situe en moyenne à 2,13% de la meilleure solution connue, tandis que la meilleure solution après 10 exécutions se situe à 0,22% pour un temps d'exécution d'environ 2,6 secondes. Quant à *ALNS*, la solution moyenne est à 0,65% et la meilleure solution à 0,05% de la meilleure solution connue pour un temps d'environ 11 secondes. Pour un temps de calcul similaire, *ALNS* supplante d'un peu plus de 3% l'algorithme de Petersen et Madsen (2007). Et pour plus de rapidité, il est possible d'obtenir une amélioration de l'ordre de 2% avec *LNS*.

Sur les instances avec 66 requêtes au tableau 5.12, l'écart moyen de *LNS* est de 5,07% pour un temps d'exécution d'environ 12 secondes alors que Petersen et Madsen (2007) obtiennent 18% en 10 secondes. *ALNS* se situe en moyenne à 2,76% en 111 secondes alors que Petersen et Madsen (2007) font 8,64% en 3 minutes.

Il apparaît donc que *ALNS* supplante l'algorithme de Petersen et Madsen (2007) autant en temps de calcul qu'en qualité de solution. Quant à *LNS*, il permet d'obtenir une qualité de solution intéressante avec des temps d'exécution réduits.

5.3.3 Résultats *TSPPDL*

Le troisième jeu d'instances provient de Carrabs, Cordeau et Laporte (2007) et de Carrabs, Cerruli et Cordeau (2007). Leur problématique consiste à construire une tournée de cueillettes et de livraisons où la livraison des colis doit respecter une contrainte de type "dernier entré, premier sorti". Ainsi, un colis i ramassé avant un colis j doit absolument être livré après j si j se retrouve au-dessus de i dans la pile. Pour transformer ce problème en $1-PDMS$, il suffit de considérer que chaque instance n'implique qu'une seule pile et que sa capacité est infinie.

Dans l'article de Carrabs, Cerruli et Cordeau (2007), les auteurs proposent une méthode exacte afin de résoudre leur *TSPPDL*. Un total de 63 instances furent conçues afin de tester leur méthode avec un nombre de sommets (incluant le dépôt) qui varie entre 19

		Petersen		*LNS*			*ALNS*		
Instance	*Optimal*	*Court*	*Long*	*Meilleure*	*Moy.*	*Temps (s)*	*Meilleure*	*Moy.*	*Temps (s)*
R00-12	694	0,00	0,00	0,00	0,61	0,3	0,00	0,52	0,5
R01-12	710	0,00	0,00	0,00	0,79	0,3	0,00	0,31	0,5
R02-12	606	0,00	0,00	0,00	0,17	0,3	0,00	0,00	0,5
R03-12	680	0,00	0,00	0,00	0,00	0,3	0,00	0,00	0,5
R04-12	607	0,00	0,00	0,00	0,21	0,3	0,00	0,00	0,5
R05-12	567	0,00	0,00	0,00	0,00	0,3	0,00	0,00	0,5
R06-12	747	0,00	0,00	0,00	0,19	0,3	0,00	0,00	0,5
R07-12	557	0,00	0,00	0,00	0,07	0,3	0,00	0,00	0,5
R08-12	690	0,00	0,00	0,00	0,43	0,3	0,00	0,00	0,5
R09-12	669	0,00	0,00	0,00	0,00	0,3	0,00	0,00	0,5
R10-12	633	0,00	0,00	0,00	0,02	0,3	0,00	0,00	0,5
R11-12	591	0,00	0,00	0,00	0,00	0,3	0,00	0,00	0,5
R12-12	722	0,00	0,00	0,00	0,33	0,3	0,00	0,17	0,5
R13-12	664	0,00	0,00	0,00	0,00	0,3	0,00	0,00	0,5
R14-12	650	0,00	0,00	0,00	0,55	0,3	0,00	0,26	0,5
R15-12	595	0,00	0,00	0,00	0,25	0,3	0,00	0,27	0,5
R16-12	577	0,00	0,00	0,00	0,10	0,3	0,00	0,00	0,5
R17-12	737	0,00	0,00	0,00	0,16	0,3	0,00	0,00	0,5
R18-12	724	0,00	0,00	0,00	0,07	0,3	0,00	0,01	0,5
R19-12	753	0,00	0,00	0,00	0,23	0,3	0,00	0,16	0,5
Moyenne		0,00	0,00	0,00	0,21	0,3	0,00	0,08	0,5

Tableau 5.10 – Résultats pour les instances à 12 requêtes de Petersen et Madsen (2007)

	Petersen			*LNS*			*ALNS*		
Instance	*Meilleure*	*Court*	*Long*	*Meilleure*	*Moy.*	*Temps (s)*	*Meilleure*	*Moy.*	*Temps (s)*
R00	1063	4,00	1,00	0,00	2,71	2,5	0,00	0,57	10,6
R01	1032	4,00	1,00	0,00	3,02	2,5	0,00	0,24	10,7
R02	1065	4,00	1,00	0,28	1,74	2,6	0,00	0,25	10,7
R03	1100	6,00	1,00	0,00	2,24	2,6	0,00	1,00	11,2
R04	1052	5,00	2,00	1,05	2,75	2,6	0,00	1,14	11,2
R05	1008	3,00	1,00	0,00	1,60	2,6	0,00	0,84	11,0
R06	1110	6,00	2,00	0,00	3,25	2,6	0,00	0,31	11,2
R07	1105	5,00	1,00	0,36	2,25	2,6	0,36	0,90	11,1
R08	1109	4,00	1,00	0,00	1,43	2,6	0,00	0,55	11,1
R09	1091	4,00	1,00	0,18	1,51	2,6	0,00	0,54	11,1
R10	1016	5,00	0,00	0,00	2,05	2,6	0,00	0,00	11,3
R11	1001	6,00	1,00	0,00	2,89	2,6	0,00	0,60	11,1
R12	1109	4,00	1,00	0,27	1,68	2,6	0,18	0,74	11,1
R13	1084	4,00	1,00	0,00	1,45	2,6	0,00	0,62	11,0
R14	1034	3,00	0,00	0,10	2,42	2,6	0,00	0,69	11,1
R15	1142	4,00	1,00	0,18	1,25	2,6	0,26	1,02	11,3
R16	1093	2,00	0,00	0,00	1,02	2,6	0,00	0,15	11,2
R17	1073	4,00	0,00	0,65	2,16	2,6	0,00	0,57	10,9
R18	1118	5,00	1,00	0,89	3,49	2,6	0,00	1,33	11,3
R19	1089	3,00	1,00	0,37	1,71	2,6	0,28	0,91	11,2
Moyenne		4,25	0,90	0,22	2,13	2,6	0,05	0,65	11,1

Tableau 5.11 – Résultats pour les instances à 33 requêtes de Petersen et Madsen (2007)

	Petersen			*LNS*			*ALNS*		
Instance	*Meilleure*	*Court*	*Long*	*Meilleure*	*Moy.*	*Temps (s)*	*Meilleure*	*Moy.*	*Temps (s)*
R00-66	1594	19,00	7,00	3,39	4,84	11,9	0,31	2,65	108,9
R01-66	1600	20,00	8,00	2,56	5,46	12,0	1,25	3,09	110,1
R02-66	1576	20,00	12,00	5,14	7,83	12,0	3,55	4,90	109,6
R03-66	1631	14,00	6,00	2,64	4,71	12,0	0,67	2,02	111,3
R04-66	1611	18,00	9,00	3,54	5,12	12,0	1,49	3,16	111,4
R05-66	1528	18,00	7,00	3,08	5,05	12,0	-0,13	1,30	112,1
R06-66	1651	17,00	7,00	0,85	4,49	12,0	0,42	2,13	111,5
R07-66	1653	17,00	8,00	3,45	4,85	12,0	1,21	3,02	111,6
R08-66	1607	18,00	7,00	2,18	4,83	12,0	0,62	3,04	111,6
R09-66	1598	18,00	8,00	2,00	4,02	12,0	2,07	2,78	112,1
R10-66	1702	17,00	9,00	0,47	3,70	11,9	0,59	1,77	112,4
R11-66	1575	19,00	8,00	1,59	4,95	12,0	0,44	2,98	111,9
R12-66	1652	19,00	10,00	1,45	4,50	12,0	0,30	1,96	112,4
R13-66	1617	19,00	10,00	2,23	4,32	12,0	1,98	3,21	111,6
R14-66	1611	21,00	9,00	4,84	6,17	12,0	0,56	2,59	111,7
R15-66	1608	19,00	10,00	2,18	4,99	12,0	1,31	2,20	111,2
R16-66	1725	16,00	7,00	1,28	4,63	11,9	0,87	2,07	111,8
R17-66	1627	21,00	10,00	2,77	5,89	12,0	2,03	3,34	112,0
R18-66	1671	18,00	8,00	4,19	6,68	11,9	1,97	3,88	112,5
R19-66	1635	17,00	9,00	1,04	4,45	12,0	1,83	3,13	112,5
Moyenne		18,25	8,64	2,54	5,07	12,0	1,17	2,76	111,5

Tableau 5.12 – Résultats pour les instances à 66 requêtes de Petersen et Madsen (2007)

et 43. De ces 63 instances, 52 instances furent résolues optimalement à l'intérieur des 3 heures requises. Pour les 11 instances restantes, les bornes supérieures trouvées par leurs algorithmes sont rapportées. La méthode exacte utilise CPLEX comme engin de résolution roulant sur un processeur AMD Opteron cadencé à 2,4 Ghz.

Les résultats pour ces instances se retrouvent au tableau 5.13. La première colonne indique le nom de l'instance, la deuxième le nombre de sommets et la troisième contient la valeur de la solution optimale (ou meilleure solution connue). Les entrées en gras sont celles où il n'existe pas de preuve d'optimalité. Les 6 colonnes suivantes contiennent la meilleure solution trouvée, la solution moyenne ainsi que le temps moyen en secondes pour les algorithmes *LNS* et *ALNS*.

Pour *LNS* et *ALNS*, les meilleures solutions obtenues sur 10 exécutions sont presque toujours égales aux meilleures solutions connues, qu'elles soient optimales ou non. L'algorithme *ALNS* obtient pratiquement toujours la meilleure solution connue avec un écart moyen de 0,056% en un temps d'exécution de moins de 1 seconde. La version *LNS* s'exécute environ 2 fois plus vite pour un écart de 0,03% supérieur à celui de *ALNS*. Il faut aussi noter qu'une nouvelle meilleure solution fut trouvée pour l'instance *nrw*1379 avec 35 sommets.

Le second article des mêmes auteurs, soit celui de Carrabs, Cordeau et Laporte (2007), propose une métaheuristique à voisinage variable (*VNS*) pour résoudre le même problème. Leur méthode utilise 8 voisinages différents pour explorer l'espace des solutions. Leurs algorithmes furent codés en C et exécutés sur un processeur Pentium 4 cadencé à 3,4 Ghz. Il faut noter que notre machine est environ deux fois plus rapide. Leur ensemble contient 42 instances et le nombre de sommets (incluant le dépôt) se situe entre 25 et 751.

Les résultats pour ces instances se retrouvent au tableau 5.14. Globalement, *ALNS* et *LNS* obtiennent en moyenne de meilleurs résultats sur toutes les instances en 3 fois et 24 fois moins de temps, respectivement. L'écart moyen avec la meilleure solution connue pour les deux algorithmes est environ 2% moindre que pour *VNS*. Toutefois, il est important de noter que la différence est très faible pour les petites instances à 25 et 51 sommets. Pour les plus grandes instances de 251 à 751 sommets, la différence est plus marquée. Ainsi, les deux algorithmes font environ 3% mieux que *VNS* pour des temps de calcul comparables. Les plus grosses différences sont observées sur les instances de 101 à 251 sommets. Par la suite, la différence diminue lorsque le nombre de sommets augmente, laissant croire qu'un plus grand nombre d'exécutions permettrait de faire mieux pour les instances ayant de 501 à 751 sommets.

Il faut noter que les meilleures solutions trouvées par les algorithmes *ALNS* et *LNS* au cours de nos expérimentations sur les instances de Carrabs, Cordeau et Laporte (2007) ont été précédemment rapportés dans la colonne "Meilleure 1 pile" du tableau 5.9.

			LNS			*ALNS*		
Instance	$2n+1$	*Optimal*	*Meilleure*	*Moy.*	*Temps (s)*	*Meilleure*	*Moy.*	*Temps (s)*
a280	19	402	0,00	0,00	0,1	0,00	0,00	0,2
	23	468	0,00	0,00	0,2	0,00	0,00	0,2
	27	505	0,00	0,00	0,2	0,00	0,00	0,3
	31	560	0,00	0,00	0,3	0,00	0,00	0,4
	35	647	0,00	0,00	0,3	0,00	0,00	0,6
	39	691	0,00	0,00	0,4	0,00	0,00	0,8
	43	752	0,00	0,00	0,5	0,00	0,00	1,0
att532	19	4250	0,00	0,00	0,1	0,00	0,00	0,2
	23	5038	0,00	0,00	0,2	0,00	0,00	0,2
	27	5800	0,00	0,00	0,2	0,00	0,00	0,3
	31	6173	0,00	0,00	0,3	0,00	0,00	0,4
	35	6361	0,00	0,00	0,3	0,00	0,00	0,6
	39	6725	0,00	0,00	0,4	0,00	0,00	0,8
	43	**10714**	0,00	0,05	0,4	0,00	0,05	1,0
brd14051	19	4555	0,00	0,00	0,1	0,00	0,00	0,2
	23	4655	1,29	1,29	0,2	1,29	1,29	0,2
	27	4936	0,00	0,00	0,2	0,00	0,00	0,3
	31	5186	0,00	0,27	0,3	0,00	0,00	0,4
	35	5196	0,00	0,00	0,3	0,00	0,00	0,6
	39	**5629**	0,00	0,00	0,3	0,00	0,00	0,7
	43	**5719**	0,00	0,00	0,4	0,00	0,00	1,0
d15112	19	76203	0,00	0,06	0,1	0,00	0,00	0,2
	23	88272	0,00	0,10	0,2	0,00	0,16	0,2
	27	93158	0,00	0,15	0,2	0,00	0,25	0,3
	31	109166	0,00	0,29	0,3	0,00	0,22	0,5
	35	115554	0,00	0,00	0,3	0,00	0,00	0,6
	39	**119863**	0,00	0,00	0,4	0,00	0,00	0,8
	43	**128798**	0,00	0,00	0,4	0,00	0,00	1,0
d18512	19	4446	0,00	0,00	0,1	0,00	0,00	0,2
	23	4658	0,00	0,00	0,2	0,00	0,00	0,2
	27	4704	0,00	1,04	0,2	0,00	0,00	0,3
	31	5120	0,00	0,00	0,3	0,00	0,00	0,5
	35	5186	0,00	0,00	0,3	0,00	0,00	0,6
	39	**5419**	0,00	0,00	0,4	0,00	0,00	0,7
	43	**5634**	0,00	0,00	0,4	0,00	0,00	1,0

			LNS			*ALNS*		
Instance	$2n+1$	*Optimal*	*Meilleure*	*Moy.*	*Temps (s)*	*Meilleure*	*Moy.*	*Temps (s)*
fnl4461	19	1866	0,00	0,00	0,1	0,00	0,00	0,2
	23	2067	0,00	0,00	0,2	0,00	0,00	0,2
	27	2483	0,00	0,00	0,2	0,00	0,00	0,3
	31	2672	0,00	0,00	0,3	0,00	0,00	0,4
	35	2852	0,00	0,00	0,3	0,00	0,00	0,6
	39	3109	0,00	0,00	0,4	0,00	0,00	0,7
	43	**3269**	0,00	0,00	0,5	0,00	0,00	1,0
nrw1379	19	2691	0,00	0,00	0,1	0,00	0,00	0,2
	23	2919	0,00	0,00	0,2	0,00	0,00	0,2
	27	3366	0,00	0,00	0,2	0,00	0,00	0,3
	31	3554	0,00	0,00	0,3	0,00	0,00	0,5
	35	**3652**	-0,22	-0,22	0,3	-0,22	-0,22	0,6
	39	**4002**	0,00	0,00	0,4	0,00	0,00	0,8
	43	**4282**	0,00	0,00	0,4	0,00	0,00	1,0
pr1002	19	12947	0,00	0,00	0,1	0,00	0,00	0,2
	23	13872	0,00	0,00	0,2	0,00	0,00	0,2
	27	15566	0,00	0,00	0,2	0,00	0,00	0,3
	31	16255	0,00	0,00	0,3	0,00	0,00	0,4
	35	17564	0,00	0,00	0,3	0,00	0,00	0,6
	39	18862	0,00	0,00	0,4	0,00	0,00	0,7
	43	20173	0,00	0,00	0,4	0,00	0,00	1,0
ts225	19	21000	0,00	0,00	0,1	0,00	0,00	0,2
	23	25000	0,00	0,00	0,2	0,00	0,00	0,2
	27	32395	0,00	0,00	0,2	0,00	0,00	0,3
	31	33395	2,06	2,06	0,3	0,00	1,65	0,5
	35	36703	0,22	0,22	0,3	0,00	0,13	0,6
	39	39395	0,00	0,00	0,4	0,00	0,00	0,8
	43	43082	0,00	0,00	0,5	0,00	0,00	1,0
Moyenne			0,053	0,084	0,3	0,017	0,056	0,5

Tableau 5.13 – Résultats pour les instances de Carrabs, Cerruli et Cordeau (2007)

		VNS		*LNS*			*ALNS*		
Instance	$2n+1$	*Moyenne*	*Temps*	*Meilleure*	*Moy.*	*Temps*	*Meilleure*	*Moy.*	*Temps*
fnl4461	25	0,00	0,0	0,00	0,00	0,2	0,00	0,00	0,3
	51	0,00	0,1	0,00	0,00	0,6	0,00	0,00	1,5
	75	2,20	0,2	0,00	0,00	1,1	0,00	0,00	4,6
	101	3,78	0,7	0,04	0,39	1,9	0,00	0,32	10,5
	251	2,51	23,9	0,12	0,96	8,1	0,00	0,66	71,5
	501	3,95	458,6	0,53	1,74	32,6	0,00	1,29	215,1
	751	3,53	2172,5	0,33	0,96	76,2	0,00	0,57	532,8
brd14051	25	0,22	0,0	0,00	0,00	0,2	0,00	0,00	0,3
	51	0,30	0,1	0,00	0,00	0,5	0,00	0,00	1,5
	75	1,07	0,3	0,00	0,00	1,1	0,00	0,00	4,0
	101	2,82	0,7	0,00	0,16	1,9	0,00	0,01	10,2
	251	8,44	36,7	1,66	3,06	8,1	0,00	1,83	72,0
	501	5,56	478,7	0,00	1,49	41,0	0,89	1,68	213,2
	751	4,63	2169,8	0,00	0,88	70,8	0,75	1,32	482,8
d15112	25	0,00	0,0	0,00	0,00	0,2	0,00	0,00	0,3
	51	1,03	0,0	0,00	0,00	0,5	0,00	0,00	1,4
	75	1,20	0,2	0,00	0,00	1,1	0,00	0,00	4,5
	101	4,41	0,5	0,00	0,64	2,0	0,00	0,36	10,8
	251	4,80	24,6	0,67	1,17	8,3	0,00	0,94	73,0
	501	3,01	385,9	0,00	0,81	35,2	0,03	1,03	218,1
	751	2,22	1968,6	0,26	0,75	64,7	0,00	0,83	465,8
d18512	25	0,24	0,0	0,00	0,00	0,2	0,00	0,00	0,3
	51	0,85	0,1	0,00	0,00	0,5	0,00	0,00	1,4
	75	1,77	0,2	0,00	0,10	1,1	0,00	0,00	3,9
	101	0,88	0,5	0,00	0,04	1,9	0,00	0,00	10,2
	251	6,94	32,9	0,00	1,20	8,0	0,15	0,99	70,2
	501	5,65	486,0	0,48	0,91	31,0	0,00	0,71	243,4
	751	3,58	2508,5	0,00	0,64	61,4	0,51	0,86	576,2
nrw1379	25	0,09	0,0	0,00	0,00	0,2	0,00	0,00	0,3
	51	0,79	0,1	0,00	0,00	0,6	0,00	0,00	1,4
	75	0,50	0,2	0,00	0,00	1,1	0,00	0,00	4,5
	101	3,88	0,5	0,00	0,56	1,8	0,00	0,34	9,9
	251	5,54	24,5	0,56	1,18	8,1	0,00	0,89	72,3
	501	3,60	380,1	0,00	0,84	31,2	0,22	1,20	218,9
	751	2,23	2447,1	0,07	0,72	61,1	0,00	0,74	516,4
pr1002	25	0,00	0,0	0,00	0,00	0,2	0,00	0,00	0,3
	51	0,81	0,1	0,00	0,00	0,6	0,00	0,00	1,5
	75	0,67	0,3	0,00	0,22	1,1	0,21	0,22	4,4
	101	3,44	0,8	0,00	0,13	1,9	0,00	0,19	10,8
	251	5,86	31,3	0,00	1,85	8,1	1,19	1,87	68,6
	501	3,57	471,9	0,85	1,34	36,7	0,00	1,25	213,5
	751	2,80	2785,4	0,00	0,86	72,6	0,04	0,49	500,7
Moyenne		2,604	402,2	0,132	0,562	16,3	0,095	0,490	117,2
Moy. ≥ 251		4,357	938,2	0,307	1,188	36,8	0,210	1,065	268,0

Tableau 5.14 – Résultats pour les instances de Carrabs, Cordeau et Laporte (2007)

CHAPITRE 6

CONCLUSION

Ce mémoire aborde un nouveau problème de génération de tournée avec contrainte de chargement qui est, en fait, une généralisation de problèmes rapportés dans la littérature. Une adaptation de l'algorithme à grand voisinage adaptatif fut proposée comme méthode de résolution. À cet effet, plusieurs voisinages d'insertion et de retrait de requêtes furent développés afin d'exploiter la structure du problème. Le nouvel algorithme s'est avéré plus performant en termes de qualité des solutions et temps d'exécution que les alternatives proposées dans la littérature. De plus, une version simplifiée de l'algorithme s'est révélée efficace sur les problèmes à une pile comme ceux de type *TSPPDL*.

Un algorithme de programmation dynamique fut conçu spécialement pour les problèmes à plusieurs piles. Celui-ci calcule la tournée optimale en fonction de l'ordre des chargements et déchargements dans chaque pile. Le gain obtenu est intéressant compte tenu du temps d'exécution rapide de l'algorithme.

Certains développements futurs peuvent maintenant être envisagés. Puisqu'il s'agit d'un premier travail sur ce sujet, il serait intéressant de concevoir d'autres métaheuristiques exploitant encore davantage la structure des piles. Dans l'algorithme *ALNS*, il n'y a pas vraiment de politique de gestion des piles, en ce sens que l'algorithme considère un seul endroit pour placer une requête. Mais il peut exister plusieurs endroits différents qui mènent à une même solution. On pourrait vouloir placer la requête sur la pile la moins remplie, afin de laisser plus de latitude lors de l'insertion des requêtes restantes. Ou encore, on pourrait vouloir favoriser des placements laissant une pile vide. Celle-ci pourrait être utilisée par la suite lors de déplacements de requêtes afin d'améliorer la solution.

Il serait également intéressant d'étudier d'autres variantes de notre problème. Ainsi, un problème où la demande d'un client se compose de plusieurs items pouvant se retrouver sur des piles différentes se rapprocherait davantage de la réalité. On pourrait aussi considérer des problèmes où les items occupent un espace en deux ou en trois dimen-

sions. Une foule de nouvelles avenues de recherche peuvent donc être envisagées pour le futur.

BIBLIOGRAPHIE

[1] Applegate, D. L., Bixby, R. E., Chvátal, V., Cook, W. J., The Traveling Salesman Problem : A Computational Study, Princeton University Press, 2006.

[2] Bräysy O, Gendreau M., Vehicle Routing Problem with Time Windows, Part I : Route Construction and Local Search Algorithms. Transportation Science 39, 104-118, 2005a.

[3] Bräysy O., Gendreau M., Vehicle Routing Problem with Time Windows, Part II : Metaheuristics. Transportation Science 39, 119-139, 2005b.

[4] Carrabs F., Cordeau, J.-F., Laporte, G., Variable Neighborhood Search for the Pickup and Delivery Traveling Salesman Problem with LIFO Loading. INFORMS Journal on Computing 19, 618-632, 2007.

[5] Carrabs F., Cerulli, R., Cordeau, J.-F., An Additive Branch-and-Bound Algorithm for the Pickup and Delivery Traveling Salesman Problem with LIFO or FIFO Loading. INFOR 45, 223-238, 2007.

[6] Cordeau J.-F., Gendreau M., Hertz A., Laporte G., Sormany J.-S., New Heuristics for the Vehicle Routing Problem. Dans : Langevin D., Riopel D., éditeurs, Logistics Systems : Design and Optimization, Sringer, 2005.

[7] Cordeau J.-F., Iori M., Laporte G., Salazar-González J.-J., A Branch-and-Cut Algorithm for the Pickup and Delivery Traveling Salesman Problem with LIFO Loading. Forthcoming in Networks (Published Online : 6 May 2009).

[8] Cordeau J.-F., Laporte G., Mercier A., A Unified Tabu Search Heuristic for Vehicle Routing Problems with Time Windows. Journal of the Operational Research Society 52, 928-936, 2001.

[9] Cordeau J.-F., Desaulniers G., Desrosiers J., Solomon M.M., Soumis F., The VRP with Time Windows. Dans : Toth P., Vigo D., éditeurs, The Vehicle Routing Pro-

blem. SIAM Monographs on Discrete Mathematics and Applications, SIAM, Philadelphia, 157-194, 2001.

[10] Dantzig G.B., Ramser J.H., The truck dispatching problem. Management Science 6, 80-91, 1959.

[11] Dell'Amico M., Monaci M., Pagani C., Vigo D., Heuristic Approaches for the Fleet Size and Mix Vehicle Routing Problem with Time Windows. Transportation Science 41, 516-526, 2007.

[12] Doerner K.F., Fuellerer G., Hartl R.F., Gronalt M., Iori M., Metaheuristics for the Vehicle Routing Problem with Loading Constraints. Networks 49, 294-307, 2007.

[13] Dueck G., Scheuer T., Threshold Accepting : A General Purpose Optimization Algorithm Appearing Superior to Simulated Annealing. Journal of Computational Physics 90, 161-175, 1990.

[14] Erdogan G., Cordeau J.-F., Laporte G., The Pickup and Delivery Traveling Salesman Problem with First-In-First-Out Loading. Computers & Operations Research 36, 1800-1808, 2009.

[15] Fuellerer G., Doerner K.F., Hartl R.F., Iori M., Ant Colony Optimization for the Two-dimensional Loading Vehicle Routing Problem. Computers & Operations Research 36, 655-673, 2009.

[16] Gendreau M., Hertz A., Laporte G., A Tabu Search Heuristic for the Vehicle Routing Problem. Management Science 40, 1276-1290, 1994.

[17] Gendreau M., Hertz A., Laporte G., New Insertion and Postoptimization Procedures for the Traveling Salesman Problem. Operations Research 40, 1086-1094, 1992.

[18] Gendreau M., Iori M., Laporte G., Martello S., A Tabu Search Algorithm for a Routing and Container Loading Problem. Transportation Science 40, 342-350, 2006.

[19] Gendreau M., Iori M., Laporte G., Martello S., A Tabu Search Heuristic for the Vehicle Routing Problem with Two-Dimensional Loading Constraints. Networks 51, 4-18, 2008.

[20] Iori M., Salazar-González J.-J., Vigo D., An Exact Approach for the Vehicle Routing Problem with Two Dimensional Loading Constraints. Transportation Science 41, 253-264, 2007.

[21] Mladenovic N., Hansen P., Variable Neighborhood Search. Computers & Operations Research 24, 1097-1100, 1997.

[22] Mester D., Bräysy O., Active guided evolution strategies for the large scale vehicle routing problems with time windows, Computers & Operations Research, 32, 1593-1614, 2005.

[23] Osman I.H., Metastrategy Simulated Annealing and Tabu Search Algorithms for the Vehicle Routing Problem. Annals of Operations Research 41, 421-452, 1993.

[24] Petersen H.L., Madsen O.B.G., The Double Travelling Salesman Problem with Multiple Stacks - Formulation and Heuristic Solution Approaches, European Journal of Operational Research 198, 139-147, 2009.

[25] Pisinger D., Ropke S., A General Heuristic for Vehicle Routing Problems. Computers & Operations Research 34, 2403-2435, 2007.

[26] Potvin J.-Y., Rousseau J.-M. An Exchange Heuristic for Routing Problems with Time Windows. Journal of the Operational Research Society 46, 1433-1446, 1995.

[27] Reimann M., Stummer M., Doerner K.F., A Savings-Based Ant System for the Vehicle Routing Problem. In : Proceedings of the Genetic and Evolutionary Computation Conference, Morgan Kaufmann, 1317-1325, 2002.

[28] Savelsbergh M.W.P., The Vehicle Routing Problem with Time windows : Minimizing Route Duration. INFORMS Journal on Computing 4, 146-154, 1992.

[29] Schrimpf G., Schneider J., Stamm-Wilbrandt H., Dueck G., Record Breaking Optimization Results using the Ruin and Recreate Principle. Journal of Computational Physics 159, 139-171, 2000.

[30] Shaw P., Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems, In : Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming, M. Maher and J.-F. Puget editors, Springer-Verlag, 417-431, 1998.

[31] Tarantilis C.-D., Kiranoudis C.T., BoneRoute : An adaptive memory-based method for effective fleet management. Annals of Operations Research 115, 227-241, 2002.

[32] Voudouris C., Guided Local Search for Combinatorial Problems. Ph.D. Dissertation, University of Essex, UK, 1997.