Master Recherche en Informatique 2009-2010

Résolution heuristique d'un problème de conception de modules automatiques de rangement

par

Paméla WATTEBLED

sous la direction de

François CLAUTIAUX

et

El-Ghazali TALBI

INRIA LILLE-NORD EUROPE, EPI DOLPHIN

Remerciements

Je tenais tout d'abord à remercier Monsieur François CLAUTIAUX, maître de conférences à IUT A de Lille 1 ainsi que Monsieur El-Ghazali TALBI, Professeur à polytech'Lille d'avoir accepté de m'encadrer et de m'avoir permis de mener à bien ce projet. Monsieur François CLAUTIAUX a toujours été présent au cours de ce stage, nous avons eu des discussions passionnantes.

Je tiens également à remercier Monsieur Jean-Luc DEKEYSER, Professeur à l'université de Lille 1 et Monsieur Nourredine MELAB, Professeur à l'université de Lille 1, tout deux responsables du master Recherche, pour la formation enrichissante qu'ils nous ont offerte et leur disponibilité.

Je remercie toute l'équipe DOLPHIN pour son acceuil chaleureux particulièrement Monsieur Bilel DERBEL et Mathieu Djamai.

Un grand merci à Malika DEBUYSSCHERE, l'assitante de l'équipe, pour toutes ses attentions. J'adresse mes remerciements à tous mes collègues de bureau: Adel, Amine, Damien et Yacine, ainsi que tous mes autres collègues de MRI notamment Mohamed Redjedal, un remerciement particulier à Nicolas Gouvy avec qui j'ai partagé ces cinq années de formation à l'université de Lille 1.

Je remercie toute ma famille, mes parents et mon frère qui m'ont soutenue et supportée. Une pensée particulière à ma petite soeur Amandine qui a pris le temps de relire ce manuscrit. Je n'oublierai pas non plus mon époux Samy et ma fille Cécilia qui me donne chaque jour beaucoup d'amour et de courage.

Résumé

Dans ce travail, nous traitons un problème d'optimisation combinatoire appliqué à un cas réel et concret. Il s'agit d'un problème de conception d'armoires automatisées pour le rangement de boîtes de médicaments, dans une pharmacie. Ainsi la pharmacie dispose d'espaces de stockage (armoires non divisées) limités, dans lesquels le flux de médicaments est géré automatiquement par un robot. L'objectif est d'utiliser de la meilleure façon possible l'espace de stockage disponible.
La résolution que nous proposons pour ce problème est constituée en deux étapes principales :

- conception initiale du module de rangement : division des armoires en étagères (séparation en largeur, profondeur et hauteur '3D');
- gestion des flux de boîtes (arrivées et sorties).

La première étape correspond à un problème classique de découpe, pour lequel on trouve plusieurs solutions dans la littérature. Cependant les solutions de la littérature pour le '3D', ne sont pas adaptées à mon problème. Nous nous sommes donc inspirés de la littérature pour les instances '2D'. Il s'agit d'heuristiques gloutonnes adaptées au contexte 3D. Elles ont été implémentées et comparées en termes de performance et de temps d'exécution.
Dans la seconde étape de notre approche, la gestion du flux de médicaments a été modélisé sous forme d'un nouveau problème : multi-sac à dos $\{0,1\}$ multipériodes avec retraits d'objets. Le but est de maximiser le profit total. Ce problème se distingue des différentes variantes du problème de sac à dos que l'on trouve dans la littérature par la prise en compte des délais de sortie des objets. Nous proposons plusieurs approches heuristiques pour résoudre ce problème particulier. Ces approches ont été comparées expérimentalement sur des données inspirées d'une application industrielle.

Abstract

In this work we deal with a combinatorial optimization problem applied to a real and concrete case. It consists in designing automated containers for storage of medicine boxes in a pharmacy. Thus, the pharmacy has limited storage space (undivided containers), in which the flow of medicines must be handled automatically by a robot. The goal is to use the available storage space in the best possible way.

The resolution that I propose for this problem consists of two main steps:

- Dividing the containers ('3 D' separation: width, depth and height)
- Management of the boxes flow (arrivals and exits)

The first step consists in the classic problem of cutting, for which there are several resolution approaches in the literature. However, the existing solutions, related to '3D ' aspects, are not suitable for the resolution of my problem. We are therefore inspired by the literature for instances '2 D ' by adapting mainly greedy heuristics. Thus, many of these approaches have been adapted to the 3D context. They are also implemented and compared, in terms of performance and execution time.

In the second stage of the approach that we propose, the medicines flow has been modeled as a problem of multi-period multi-dimensional knapsack with objects removal. The goal is to maximize the total profit of the pharmacy. This new problem differs from classical knapsack problems found in the literature by taking into account the time of object release. Thus we have proposed several heuristics to solve this particular problem. These approaches were compared in an experimental way based on data from real industrial application.

Table des matières

Chapitre 1

Introduction

Contexte et motivation

L'évolution rapide des puissances de calcul lors des dernières années a permi d'aborder et de résoudre des problèmes d'optimisation réputés d'une grande difficulté. Ceci a également contribué à l'intérêt grandissant pour les techniques de recherche opérationnelle dans des domaines de plus en plus variés. En effet, la course vers la compétitivité, les performances et la rentabilité rendent les méthodes d'optimisation indispensables dans tous les secteurs économiques.

Le problème traité dans ce manuscrit correspond à un besoin réel dans le domaine de la robotique pour les pharmacies, et a été fourni par l'entreprise INTECUM (Bastia). Il consiste à concevoir des armoires, pour y ranger de façon optimisée des boites de médicaments de diverses tailles. Les arrivées et les sorties de boites sont dynamiques (évolutives dans le temps), et leur placement dans les armoires est réalisé automatiquement par un robot.

Il s'agit d'un problème dit difficile ou NP-Complet car il ne peut être résolu (de façon générale), de façon exacte et optimale, en un temps raisonnable. Par conséquent,les approches basées sur des heuristiques sont de bons candidats pour résoudre ce problème. La première classe consiste en des méthodes basées généralement sur l'intuition. Le but dans ce genre d'approches est d'aboutir à de bonnes solutions, en réglant certains paramètres. Quant à la seconde, ce sont des procédures approximatives avec de bonnes méthodes de recherche à travers l'espace de solutions. La rapidité de convergence vers une très bonne solution, constitue leur atout majeur.

De nos jours il n'existe aucune approche dans la littérature permettant la résolution du problème qui nous a été présenté. Ainsi, afin de l'aborder de façon efficace avec des approches heuristiques et méta-heuristiques, nous l'avons séparé en deux sous-problèmes :

- conception des conteneurs : consistant à trouver un découpage optimisé (en largeur,

longueur et hauteur) des conteneurs disponibles pour le stockage des boites de médicaments;

- placement (rangement) des boites dans les armoires, en prenant en compte leur dimensions, dates d'arrivées, dates de sorties ainsi que les bénéfices qu'elles génèrent.

L'avantage principal de ce découpage du problème est la possibilité de modéliser chacun des sous problèmes sous forme d'un problème d'optimisation. Cela nous apporte d'une littérature très riche.

Objectifs

L'objectif principal de cette conception est de fournir une approche de résolution, efficace et rapide, du problème global. Cependant, pour atteindre ce but, il faut apporter des réponses aux questions et problématiques intermédiaires suivantes :

- Quelle modélisation est efficace pour le problème ?
- Quelles méthodes et algorithmes sont performants pour résoudre le problème ?

Contributions

Les contributions de ce stage sont de diverses natures. En effet elles se déclinent sous forme de modélisation, développement, étude de l'existant, proposition de nouvelles approches ainsi que des analyses et comparaisons. Elles peuvent être résumées comme suit :

- modélisation du problème se rapprochant de modèles classiques (découpage et multi sac à dos multi-périodes)
- etude des approches de résolution existantes pour les deux sous problèmes modélisés.
- adaptation des heuristiques de résolution 2D du problème de découpe au contexte 3D.
- implémentation des approches 3D obtenues ainsi que l'analyse et la comparaison de leurs performances.
- étude de l'existant sur les différents problèmes de sac à dos
- définition, implémentation et comparaison de différentes méthodes de résolution pour le problème du multi sac à dos multi-périodes avec retrait d'objets.

Plan du manuscrit

Le chapitre 2 présente plus en détails notre problème et introduit notre méthode de résolution. Une présentation et une analyse des heuristiques guillotine existantes pour les problèmes de découpage en 2D et 3D ainsi que la présentation des différents problèmes de sac à dos, sont fait dans la chapitre 3. Le chapitre 4 est consacré à l'adaptation des heuristiques 2D aux problèmes de paking 3D. Il donne un aperçu de l'implémentation de nos algorithmes ainsi qu'une évaluation de ceux-ci. Le chapitre 5 présente les différentes versions que nous avons développées pour résoudre le problème muti-sac à dos multi-période avec retrait, dans cette section nous ferons également une évaluation et comparaison des différentes propositions. Le chapitre 6 conclut ce document et donne de multiples perspectives.

Chapitre 2

Problème de conception, approche de résolution

2.1 Problème de conception

Le problème qui nous amène à ces travaux est un problème de conception de modules de rangement automatique. Il s'agit de différentes armoires, ou modules de rangement de boîtes à médicaments. Un robot doit gérer les entrées et sorties de médicaments tout en maximisant les gains (le revenu des ventes) au cours du temps. Ce problème est complexe et il n'a à ma connaissance jamais été modélisé formellement.
Ainsi deux solutions s'offraient à nous : le modéliser mathématiquement et définir une toute nouvelle approche (à partir de zéro) au problème ainsi obtenu ou le diviser en sous problèmes pouvant être modélisés et résolus avec des formalismes et méthodes connus (avec éventuellement quelques modifications).
La première solution nous semble à proscrire en raison de la complexité du problème mais aussi parce qu'elle nous priverait de réutiliser des modèles et algorithmes ayant une efficacité prouvée depuis des années.
Nous avons donc décomposé la résolution du problème en trois parties. La première phase va être de structurer les armoires de rangement (placer des étagères). La seconde est le pilotage du robot dans le temps (entrées et sorties de boîtes) en maximisant les poids. Quant à troisième phase qui ne sera pas traitée dans ce manuscrit, elle consiste à pouvoir revenir à la première phase pour restructurer les modules en fonction des informations obtenues avec différentes simulations (Voir Fig. 2.1).

La première étape consiste donc à essayer de placer des boîtes rectangulaires dans un nombre minimum de conteneurs en minimisant l'espace perdu. Nous voulons ainsi ranger des boîtes de médicaments dans différents modules.
Pour cela, les boîtes ont une orientation fixée, de telle sorte que leur longueur soit supérieure à la largeur qui elle même sera supérieur à la hauteur : $L \geq W \geq H$ (voir Fig.2.2). Dans ce problème on dispose de différents types de boîtes (L,H,W différents).

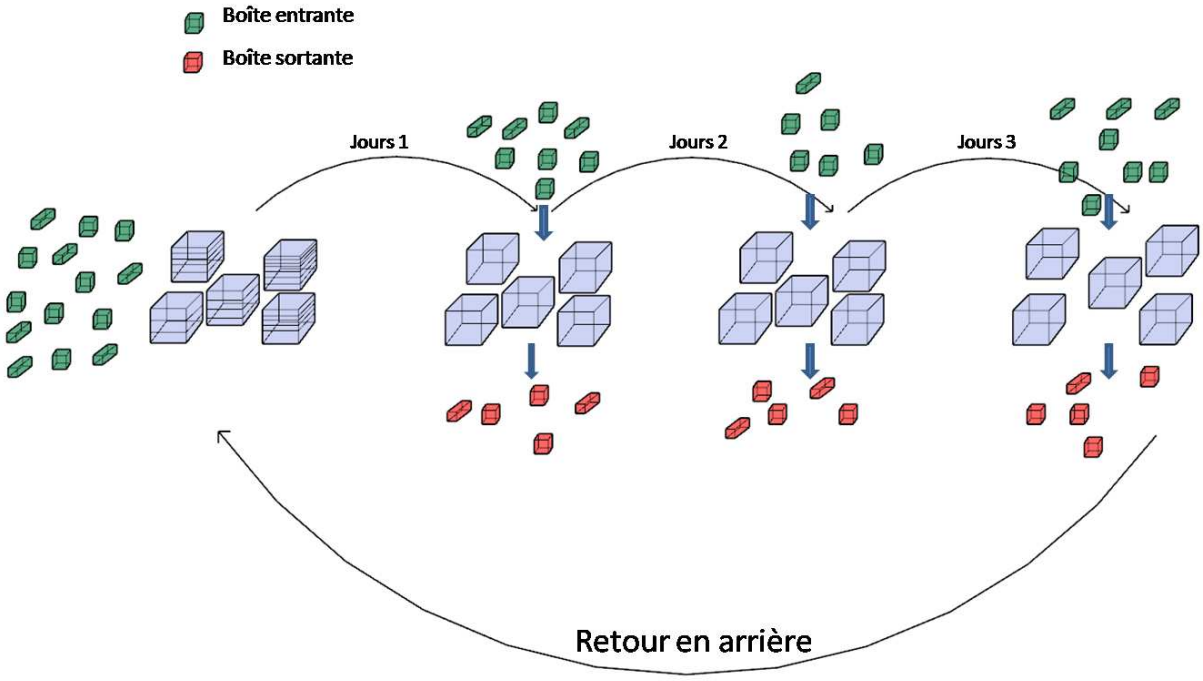

Figure 2.1: Les différentes phases de conception

Les modules sont de taille fixe et peuvent être conçus de deux façons. La première étant à l'horizontale par différentes étagères, et la seconde à la verticale dans chaque étagère. Cette dernière découpe donne différents casiers sur une même étagère (Voir Fig. 2.3), et chaque casier sera rempli d'une seule rangée de médicaments (Voir Fig. 2.4).
Dans la suite du document nous considèrons que tous les conteneurs sont de même dimension. L'objectif est de minimiser le nombre de conteneurs pour ranger un ensemble de boîtes de médicaments (minimiser la perte d'espace).

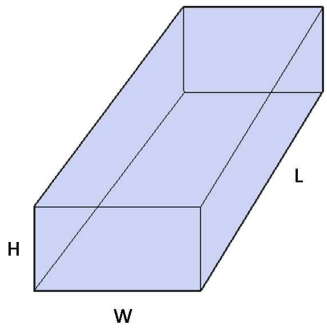

Figure 2.2: Une boîte

Le second objectif est de maximiser les gains (ventes) au cours du temps. Nous considérons que chaque type de boîte posséde un prix fixe et une durée de vie limitée dans le conteneur (correspondant à une vente). L'idéal serait donc de pouvoir ranger un

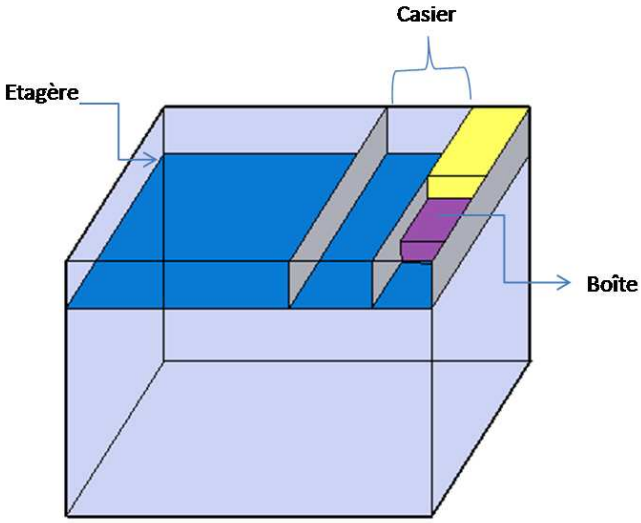

Figure 2.3: Exemple de conteneurs

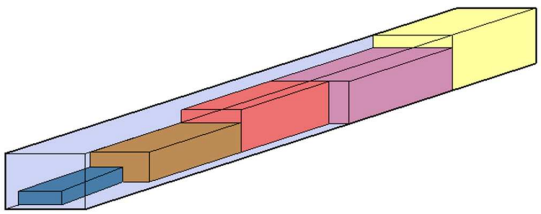

Figure 2.4: Exemple d'une rangée

maximum de petites boites, avec un prix élevé et une très courte durée de vie dans le conteneur. Ainsi l'on pourra renouveler très vite les boîtes du conteneur, en ranger un maximum pour avoir le gain le plus important possible.

2.2 Problèmes de sac à dos$\{0,1\}$ multi-période avec sortie des boîtes

Dans cette partie, nous décrivons les différentes variantes de problèmes de sac à dos qu'il serait intéressant de résoudre dans le cadre de nos travaux sur le problème de packing dynamique.

2.2.1 Versions déterministes

On notera l'horizon de temps $1, \ldots, T$. La taille du sac à dos est C. Pour chaque article i, on note t_i sa date d'arrivée, v_i sa date de vente, c_i sa taille et p_i son poids.

2.2.1.1 Un seul sac, sans réoptimisation

Dans cette variante, on peut acheter ce que l'on veut, quand on veut (il faut cependant respecter la contrainte de sac à dos). Connaissant toutes les ventes par avance, il suffit de résoudre le problème de sac à dos relatif à chaque jour de vente (on ne fait pas de réserve).

2.2.1.2 Un sac, $\{0,1\}$

Les seules variables de décision sont les x_i pour chaque i : détermine si on accepte ou non l'article lors de sa date d'arrivée. On notera I^t l'ensemble des articles tels que $t_i \leq t < v_i$.

$$\max \sum_{i \in I} p_i x_i \tag{2.1}$$

$$s.t. \sum_{i \in I^t} x_i c_i \leq C, t = 1, \ldots, T \tag{2.2}$$

$$x_i \in \{0,1\}, \forall i \in I \tag{2.3}$$

2.2.1.3 Multi-sac, $\{0,1\}$, sans réoptimisation

Dans cette version, on considère qu'un article rangé dans un sac ne peut pas changer de position entre deux unités de temps.

Dans le cadre du multi-sac, on suppose que chaque sac j a une taille C_j. On note le nombre de sacs M.

Cette fois, on introduit des variables x_{ij} qui déterminent non seulement si on entre i dans les sacs, mais aussi dans quel sac.

$$\max \sum_{i \in I} \sum_{j=1\ldots,M} p_i x_{ij} \tag{2.4}$$

$$s.t. \sum_{i \in I^t} x_{ij} c_i \leq C_j, j = 1, \ldots, M, t = 1, \ldots, T \tag{2.5}$$

$$x_{ij} \in \{0,1\}, \forall i \in I, j = 1, \ldots, M \tag{2.6}$$

2.2.1.4 Multi-sac, $\{0,1\}$, avec réoptimisation

Dans cette version, on considère le problème où l'on peut réoptimiser les sacs, c'est à dire changer la position des articles déjà rangés.

On introduit des variables x_{ijt} qui déterminent où est i au temps t, en plus des variables x_i initiales.

$$\max \sum_{i \in I} p_i x_i \tag{2.7}$$

$$s.t. \sum_{i \in I^t} x_{ijt} c_i \leq C_j, j = 1, \ldots, M, t = 1, \ldots, T \tag{2.8}$$

$$\sum_{j=1,\ldots,M} x_{ijt} = x_i, \forall i \in I, t = t_i, \ldots, v_i - 1 \tag{2.9}$$

$$x_{ij} \in \{0, 1\}, \forall i \in I, j = 1, \ldots, M \tag{2.10}$$

2.2.1.5 Plusieurs sac, sans réoptimisation

Même problème que précédemment avec plusieurs sacs. Dans ce cas, quand on sort un type d'article, on doit en plus décider lequel.

2.2.2 Versions stochastiques

Chacun des problèmes précédents peut être traité de manière stochastique. On supposera que les ventes suivent une loi de Poisson et que l'approvisionnement est corrélé aux ventes. Plusieurs versions peuvent être étudiées : quand on connaît les ventes et les approvisionnement, mais pas leur date d'arrivée, ou quand on n'a qu'une estimation des approvisionnements.

2.2.3 Bilan

Nous assimilons notre problème au modèle Multi-sac, $\{0, 1\}$, sans réoptimisation. Sachant que le problème avec la réoptimisation est tout à fait envisageable, mais le temps ayant manqué, nous traitons ici sans réoptimisation.

Chapitre 3

Etat de l'art

3.1 Introduction

Le problème d'emballage est un problème qui a été largement traité dans la littérature. Il est donc important de s'y reporter et ainsi pouvoir faire un tri de ce qui est possible d'utiliser dans notre cas. Dans cette section nous faisons un état de l'art de l'existant sur les problèmes de packing en 2D puis en 3D.
Dans la suite de la section nous présentons certains travaux représentatifs sur les problèmes de sac à dos sous ces différentes formes (un sac, multi-sac, sac multi-période).

3.2 Bin Packing

3.2.1 Algorithmes existants pour les problèmes de Strip packing en 2D

Nous avons une collection de rectangles et un conteneur 2D illimité en hauteur.

3.2.1.1 L'algorithme NFDH (Next-Fit Decreasing Height)

Dans l'algorithme NFDH [5], les rectangles sont triés selon leur hauteur de façon décroissante. On dépose en bas à gauche le premier rectangle (le plus haut) qui détermine la hauteur de l'étage (définitivement fixé). Puis l'on dépose à coté de celui-ci un autre rectangle (qui est le plus haut rectangle parmi le reste), et l'on continue jusqu'à ce que le rectangle courant ne rentre plus par sa largeur. Dans ce cas, nous déposons ce rectangle au dessus à gauche de l'étagère en créant un nouvel étage.
Cette opération est réitérée jusqu'à ne plus avoir de rectangle, Voir Fig. 3.1.

3.2.1.2 L'algorithme FFDH (First-Fit Decreasing Height)

Le principe de FFDH [5] est très similaire à celui de NFDH. Cependant avant de placer le prochain rectangle sur le niveau courant, nous essayons de le placer dans le niveau inférieur (le plus bas possible), ayant suffisamment d'espace pour l'acceuillir.
Dans FFDH, nous pouvons donc revisiter un niveau inférieur, ce qui n'est pas permis dans NFDH, Voir Fig. 3.1..

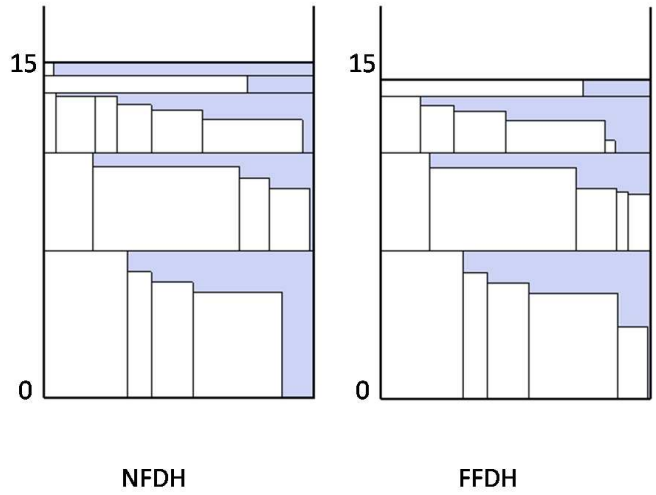

Figure 3.1: NFDH et FFDH

3.2.1.3 L'algorithme BFDH (Best-Fit Decreasing Height)

L'algorithme BFDH [10] agit de la même façon que FFDH, sauf que la recherche d'un emplacement du rectangle courant se fait de façon exhaustive sur les étagères. Ainsi l'espace horizontal perdu par le placement d'un rectangle est calculé pour toutes les étagères. Le rectangle est effectivement placé dans l'étagère pour laquelle cet espace perdu est minimal.
On peut ainsi logiquement s'attendre à ce que cet algorithme soit plus lent que FFDH qui place les rectangle dans la première étagère trouvée.

3.2.1.4 L'algorithme KP01 (knapsack)

L'algorithme KP01 est décrit ici dans le contexte particulier d'emballage de bandes. Les rectangles sont ordonnés de façon décroissante par rapport à leur hauteur. Chaque niveau est initialisé par un rectangle d'emballage avec la plus grande hauteur parmi tous les rectangles déballés. Après cette phase d'initialisation, le problème de sac à doc est résolu en maximisant le volume dans chaque niveau et vérifiant que la somme de toutes les largeurs des boîtes d'un même niveau est inférieure ou égale à la largeur totale.
En ce qui concerne notre problème qui est de minimiser le nombre de containers; Andrea Lodi, silvano Martello et Daniele Vigo [2] proposent un algorithme basé sur le KP. Celui-ci agit comme le KP précédement décrit, mais après il récupére toutes les étagères de

hauteurs h_i, et termine par résoudre avec ces différents niveaux un problème de Bin Packing à une dimension.

3.2.1.5 Variantes de l'algorithme de NFDH

Deux variantes possibles de l'algorithme NFDH proposées par Ntene [8], soit l'algorithme NFDHIW (Next-Fit Decreasing Height Increasing Width) ou l'algorithme NFDHDW (Next-Fit Decreasing Height Decreasing Width). Ces algorithmes sont similaires à l'algorithme original, la seule différence étant que pour la même hauteur, le tri est fait de façon croissante (respectivement décroissante) par rapport à la largeur.

3.2.1.6 Variantes de l'algorithme de FFDH

Les deux variantes proposées par Ntene [8] de l'algorithme FFDH sont l'algorithme FFDHIW(First-Fit Decreasing Height Increasing Width) et FFDHDW(First-Fit Decreasing Height Decreasing Width). Le principe est le même que FFDH, mais pour la même hauteur et comme les variantes de NFDH, nous faisons le tri de façon croissante (respectivement décroissante) par rapport à la largeur.

3.2.1.7 Variantes de l'algorithme de BFDH

De même que pour les algorithme NFDH et FFDH, nous pouvons faire les même variantes sur l'algorithme BFDH toujours proposées par Ntene [8], ce qui donnera les algorithmes BFDHIW(Best-Fit Decreasing Height Increasing Width) et BFDHDW(Best-Fit Decreasing Height Decreasing Width).

3.2.1.8 L'algorithme de HFF(Hybrid First-Fit)

Cet algorithme a été proposé par Chung et al.[6]. L'objectif de l'algorithme étant de faire rentrer des boites rectangulaires dans un minimum de conteneurs. Dans une première phase, l'algorithme FFDH est effectué pour créer une collection de bandes de diffèrentes hauteurs. Puis par heuristique, ils emballent les différentes bandes dans les conteneurs.
Cet algorithme a une grande importance pour nous, puisque nous voulons également minimiser les conteneurs. Cette deuxième phase est identique à la deuxième phase de l'algorithme KP01 présenté un peu plus haut dans la section.

3.2.2 Algorithmes existants pour les problèmes de bin packing en 3D

Les algorithmes que nous allons voir dans cette section sont basés sur la démarche de construction de murs. C'est à dire que le conteneur donné est rempli en couches verticales successives (respectivement horizontale) dans la direction longitudinale(respectivement latitudinale) du conteneur, voir Fig.3.2.

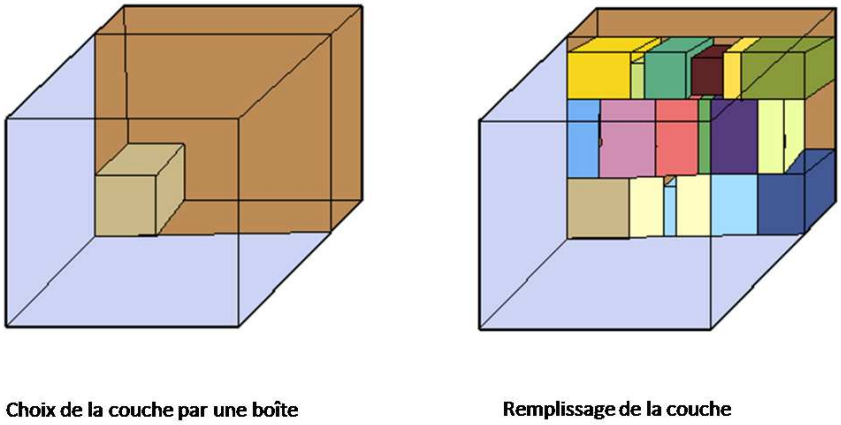

Figure 3.2: Exemple de coupe

3.2.2.1 L'algorithme de George et Robinson

Dans l'algorithme de George et Robinson [7], chaque couche (verticale) est divisée en plusieurs bandes horizontales superposées les unes sur les autres. Une seule bande est constituée de plusieurs cases placées de manière séquentielle et parallèle à la largeur du conteneur. La qualité de la solution dépend principalement de la sélection de la profondeur de la couche.

3.2.2.2 L'algorithme de Pisinger

L'algorithme proposé par Pisinger [4] a une démarche de construction de murs où un algorithme d'arbres de recherche est utilisé pour trouver la collection de la profondeur des couches et des largeurs de bandes ayant pour résultat le meilleur remplissage global. Ne pouvant pas examiner toutes les profondeurs de la couche et les largeurs de bandes (cela serait trop couteux). Pour diminuer la complexité Pisinger utilise une approche m-cut, où seul un nombre fixe de sous noeuds est considéré pour chaque noeud de branchement. Pour chaque branche, l'on considère M_1 différentes couches de profondeurs choisies selon une règle spécifiée. Chaque couche est ensuite remplie avec différentes bandes (orientées horizontalement ou verticalement).

Seul un nombre limité M_2 de largeurs de bande différentes est envisagé, pour éviter les pertes d'espaces ou les boîtes trop grandes.

Une fois la profondeur de la couche et la largeur de bande réglées, on remplit chaque profondeur de couche $d_1 ... d'_{M1}$, puis on choisit la couche de remplissage pour laquelle on a le meilleur résultat (soit la meilleure utilisation globale du volume). Puis l'on passe à la couche suivante.
Une fonction récursive est utilisée pour sélectionner les M_1 différentes couches de profondeurs, puis pour tout l'ensemble des profondeurs de $d'_1 ... d'_{M1}$ on appelle une autre

fonction qui est chargée de remplir les couches. La fonction qui doit remplir les couches le fait de façon à remplir chaque bande à l'intérieur de façon optimale comme un KP (Knapsack Problem).

- le choix de profondeurs et de largeurs
  À chaque noeud de branchement on considère les M_1 les mieux classés parmi les profondeurs, ou les M_2 les mieux classé (largeurs/hauteurs) d'une bande. Les expériences ont montré que les performances de l'algorithme reposent strictement sur le classement de ces dimensions, donc plusieurs fonctions de classement ont été étudiées. Toutes les fonctions de classement sont basées sur quelques statistiques sur les dimensions des cases restantes.

- Remplissage d'une simple bande
  Les bandes peuvent être remplies horizontalement ou verticalement. Ce remplissage est fait de façon optimale. En effet, la recherche exhaustive et optimale est tout à fait envisageable pour remplir une bande car l'ensemble des solutions restent raisonnable.

- Le jumelage des boîtes
  Pour une profondeur donnée, on peut choisir deux boîtes a et b de telle sorte que $d_a + d_b \leq d$'. Cependant, ceci ne pourra pas être envisagé dans notre problème puisque nous ne voulons que des rangées avec des boîtes les unes à la suite des autres et non deux boîtes à côtés.

  A. Bortefeldt [1] a proposé une variante à l'algorithme de D. Pisinger. Celle-ci consiste à adapter l'algorithme pour le rendre compatible dans le cas de conteneur avec une dimension variable (conteneur non fermé).

3.2.3 Algorithmes que nous ne pourrons pas prendre en compte

Cette section aborde également l'état de l'art des problèmes d'emballages. La différence étant que nous traitons là les algorithmes que nous ne pouvons pas utiliser dans le cas qui nous concerne.

3.2.3.1 Algorithmes pour les problèmes de packing en 2D non adaptés

L'algorithme SF (Split Fit) et l'algorithme de FCNR (Floor-Ceiling No Rotation)

Les 2 algorithmes, SF[5] et FCNR[3], ne peuvent pas être appliqués pour ce que nous voulons faire, puisqu'ils permettent de mettre plusieurs boîtes les unes au dessus des autres pour un même niveau.

SAS(Size Alternating Stack)

L'algorithme FFDH est connu pour avoir de mauvaises performances lorsqu'il y a une grande différence entre les hauteurs des objets d'un même niveau. Cette observation a motivé la création d'un nouvel algorithme SAS (Size Alternating Stack) [8]. Dans cet algorithme deux listes sont créées à partir de la liste initiale des bandes. La première liste est celle des bandes dont la hauteur est supérieure à la largeur, la seconde liste est celle dont la largeur est supérieure à la hauteur. Les rectangles de la liste 1 (resp. 2) sont classés de façon décroissante par rapport à leur hauteur (resp. largeur). Si un rectangle est carré, on l'insère dans la liste 2. L'idée principale de cet algorithme est d'alterner entre les deux listes de rectangles étroits et larges. Chaque niveau est initialisé en comparant les hauteurs des 2 listes et en prenant la hauteur max.

Une fois qu'une liste a été identifiée, les rectangles de cette liste en particulier, sont empilés les uns sur les autres, à partir de la limite inférieure d'un niveau jusqu'à la limite supérieure.

Lors de l'empilage des rectangles de largeur, il peut arriver que les largeurs des rectangles ultérieures ne soient pas égales, une zone rectangulaire vide apparaît dans ce cas. Ces régions rectangulaires sont utilisées pour emballer des rectangles étroits.

3.2.3.2 Algorithmes pour les problèmes de packing en 3D non adaptés

L'algorithme de Pisinger

Comme nous avons pu le voir un peu plus haut dans la section, nous pouvons nous inspirer de l'algorithme de Pisinger. Cependant nous ne pouvons pas tout utiliser dans cet algorithme. Nous avons déjà dis précédemment que nous ne pouvons pas utiliser l'aspect de jumelage des boîtes puisque nous voulons avoir une seule rangée de boîtes dans chaque boîte. Mais Pisinger utilise également un arbre de recherche pour déterminer les meilleures orientations possibles des boîtes. Dans le problème que nous résolvons dans ce travail nous voulons une orientation fixe des boîtes, ainsi que des bandes.

3.2.4 Bilan de l'état de l'art sur le conditionnement

La littérature propose différentes solutions pour notre problème, mais aucune d'elles n'est applicable directement, à notre cas. Les algorithmes en 2D de l'état de l'art doivent être modifiés pour passer en 3D et changer l'objectif (minimiser les nombres de conteneurs). En ce qui concerne les algorithmes en 3D, nous devons également en proposer des variantes. En effet les algorithmes de Pisinger et Bortefeldt ne sont pas adaptés à notre problème, ils permettent les rotations de boîtes mais également des couches horizontales et verticales dans un même conteneur, ce qui est impossible dans le problème posé. Ce que l'on retiendra principalement de cet état de l'art est la richesse à la fois qualitative (performances des algorithmes) que quantitative (diversité des approches et des variantes). Ainsi, même si aucun de ces travaux ne répond directement à la problèmatique qui nous

intéresse, on considère tout de même que l'on part d'une base existante et non de zéro (from scratch). En effet, certains des algorithmes présentés sont modifiés et adaptés ainsi au contexte 3D pour devenir utilisable lors de notre première phase de résolution.

3.3 Problème du sac à dos

3.3.1 Problème du sac à dos (KP)

Le problème de sac à dos est un problème courant en recherche opérationnelle largement étudié au cours du siècle passé. Le problème de sac à dos a été modélisé la première fois par George Ballard Mathews en 1897 [12]. Il a de multiples applications pratiques telles que le chargement de conteneurs, les découpes de matériaux...
Le problème de sac à dos a une formulation simple et il est NP-difficile. Il faut maximiser les gains rentrés dans un sac à dos sachant que chaque objet à un volume et un gain. Le sac à dos est limité en taille et ne peut donc pas contenir tous les objets. Quels objets choisir?

3.3.2 Problème du sac à dos multidimensionnel(MKP)

Le problème MKP est une variante du problème de sac à dos qui a été également étudié à de nombreuses reprises. On trouve une pléthore d'heuristiques gloutonnes dans la littérature pour résoudre ce problème. Leur quasi-totalité est basée sur un principe très simple. Elles sélectionnent les objets un par un selon un critère donné et en assurant une solution correcte. Quand un objet est dans le sac, il reste à l'intérieur, pas de possibilité de retour en arrière.
Une autre approche possible pour résoudre le MKP est les métaheuristiques, cette approche est plus récente, mais se développe bien. Des algorithmes basés sur la recherche tabou ont été proposé, par exemple par F.Glover et G.A Kochenberger [13] ou encore S.Hanafi et A.Fréville [14].

3.3.3 Problème multi-sac à dos multipériode (GMPMKP)

Le problème GMPMKP, est une variante assez récente du problème de sac à dos, il s'agit de remplir un sac à dos dans le temps. Des objets arrivent à des instants différents et il s'agit alors de les accepter ou non.
Dans la littérature que nous avons parcouru, il n'existe que des méthodes stochastiques [15]. Chaque objet est associé à une probabilité (pouvant être modifiée par apprentissage [16]), à partir de celle-ci et d'autres critères la décision est prise de mettre l'objet dans le sac. Quand un objet est déposé dans un sac, il reste définitivement à l'intérieur.

3.3.4 Bilan de l'état de l'art sur le problème de sac à dos

Le problème de sac à dos comporte de nombreuses variantes et de nombreuses applications. Néanmoins en étudiant la littérature aucune variante pouvant être assimilée à notre problème n'a été trouvée. En effet la possiblité de retrait d'objet n'a pas été prise en considération dans les travaux disponibles. Néanmoins cette étude bibliographique nous permet aujourd'hui d'envisager la modélisation de la seconde phase de notre problème comme une nouvelle variante du sac à dos. Cette modélisation permet d'obtenir un formalisme bien connu depuis des dizaines d'années. Cependant, nous ne pouvons malheureusement appliquer des approches de résolution existantes.

Chapitre 4

Adaptation des algorithmes heuristiques 2D aux problèmes de packing 3D

4.1 Introduction

Dans cette section nous allons voir comment adapter les différentes heurisitiques de la littérature du 2D à notre problème en 3D. Nous détaillons les différents algorithmes et donnons leur pseudo-code. A la fin de la section nous comparons et analysons les performances des différents algorithmes.

4.2 L'algorithme NFDH adapté à l'emballage 3D

Cet algorithme classe les différentes boîtes selon leur hauteur, puis les range dans l'ordre décroissant. Quand une boîte ne rentre plus dans la couche donnée par sa largeur on la monte sur un étage au dessus, voir Fig. 3.1.
Cet algorithme est facilement adaptable à notre problème 3D. En effet, nous adoptons le même principe en classant les boîtes par hauteur décroissante. Après avoir placé la première boîte au fond à gauche de notre conteneur nous faisons une coupe, qui va déterminer la hauteur (hauteur de la boîte sélectionnée) de l'étagère puis une seconde coupe qui va déterminer la largeur d'une rangée (largeur de la boîte sélectionnée) dans l'étagère. Après avoir obtenu une rangée, il faut la remplir de la même façon que NFDH le fait en 2D, néanmoins nous avons fait une sélection par la hauteur. Donc il se peut qu'une boite plus petite en hauteur que celle sélectionnée soit plus large. Ainsi nous devons filtrer notre liste de boîtes qui est décroissante par la hauteur pour enlever celle dont la largeur serait supérieure à la largeur de notre rangée. Ce filtrage n'est pas obligatoire, mais s'il n'est pas effectué, très rapidement une boîte ne pourra pas se placer, et donc les performances de NFDH risque de se dégrader fortement.

Après avoir fait cette initialisation nous allons faire la même chose que NFDH, en ajoutant à la suite des boîtes jusqu'à ce que la prochaine ne puisse plus entrer dans la rangée. Dans ce cas là, une autre bande est créée avec la boîte ayant la plus grande hauteur parmi toutes les boîtes y compris celles que l'on avait filtré. De nouveau, un filtre sera fait pour cette nouvelle tranche. Nous continuons l'algorithme sur toute la couche, à la fin de la couche, nous en reconstruisons une nouvelle au dessus. Et à la fin du remplissage du conteneur, nous continuons avec un autre conteneur, jusqu'à ce que toutes les boîtes soient rangées.

Nous avons implémenté l'algorithme NFDH et je donne ici le pseudo-code correspondant (Algorithme 1).

Algorithm 1 NFDH

Require: List *boites* , List *conteneurs*

```
1: conteneurCourant, etagereCourante, caseCourante
2: conteneurs.ajouteConteneur()
3: conteneurCourant=conteneurs[0]
4: boites.trierParHauteurDecroissante()
5: for i = 0 to boites.size() do
6:    boiteCourante = boites[i]
7:    {première boite que l'on doit placer, aucune case existe}
8:    if conteneurCourant.pasEtagere() then
9:       conteneurCourant.ajouteEtagere(boiteCourante.hauteur())
10:      etagereCourante=conteneurCourant.derniereEtagere()
11:      etagereCourante.ajouteCase(boiteCourante.largeur())
12:      caseCourante= etagereCourante.dernieCase()
13:   end if
14:   if caseCourante.peutContenir(boiteCourante) then
15:      caseCourante.ajoute(boiteCourante)
16:   else if etagereCourante.peutContenir(boiteCourante) then
17:      etagereCourante.ajoute(boiteCourante)
18:   else if conteneurCourant.peutContenir(boiteCourante) then
19:      conteneurCourant.ajoute(boiteCourante)
20:   else
21:      conteneurCourant=creerConteneur(boiteCourante)
22:      conteneurs.ajoute(conteneurCourant)
23:   end if
      {Faire les mises à jours de caseCourant, etagereCourante}
24: end for
```

4.3 L'algorithme FFDH adapté à l'emballage 3D

Cet algorithme est très facile à adapter à partir du moment où l'on a déjà une variante de NFDH pour les problèmes de bin packing en 3D. En effet, lorsque l'on doit placer une boîte, il faut regarder dans les couches précédentes en commençant par la plus basse

possible pour essayer de trouver une place potentielle à notre boîte. Bien sûr dans une même couche il est possible d'avoir différentes possibilités de placement de la boîte. La boîte peut être ranger dans la bande la plus ancienne (où une place est libre) pour respecter le plus possible l'algorithme FFDH. Mais ce n'est pas une obligation, et il est possible à l'intérieur d'une couche d'utiliser une autre règle de placement.

J'ai codé deux versions de l'algorithme FFDH (Algorithme 2), la seconde version (FFDHBIS) a été faite pour diminuer la complexité, donc le temps d'exécution. Au lieu de placer directement les cases créées dans les conteneurs. Nous créons d'abord une collection de cases avec les boîtes. Après avoir obtenu cette liste (collection de cases), nous allons créer des étagères avec des cases, puis nous allons ranger les étagères dans les conteneurs.

4.4 L'algorithme BFDH adapté à l'emballage 3D

L'algorithme BFDH agit de la même sorte que FFDH, sauf qu'au lieu de placer la boîte dans la couche la plus inférieure, elle est placée dans la couche qui laisse le moins d'espace horizontal.
Quatre variantes sont donc possibles, soit on insère ce nouvel élément dans la bande qui laisse le moins d'espace possible au niveau de la hauteur, de la profondeur, de la largeur ou encore celle qui laisse le moins d'espace volumique.

Pour l'algorithme BFDH, deux versions ont également été implémentées, avec les mêmes différences que l'algorithme FFDH. Dans ce document nous donnons le pseudo-code de la seconde version algorithme BFDH2 (Algorithme 3).

4.5 Les variantes aux algorithmes NFDH, FFDH et BFDH adaptés à l'emballage 3D

Les variantes des algorithmes FFDH, NFDH et BFDH peuvent très bien être adaptées dans notre cas et bien sûr, nous pouvons rajouter 2 autres variantes aux 2 proposées, par Ntene, pour chaque algorithme. En effet les variantes trient les boîtes de même hauteur dans l'ordre décroissant ou croissant selon la largeur. En passant à la 3 ème dimension, nous pouvons également trier les boîtes de même hauteur selon la largeur, mais également la profondeur (croissante ou décroissante).

4.6 L'algorithme KP01 adapté à l'emballage 3D

L'algorithme KP01 peut être adapté sans difficulté à notre problème. Nous devons ordonner de façon décroissante les boîtes par rapport à leur hauteur. Comme pour le 2D, on initialise une couche par une boîte. Puis nous remplissons de façon optimale chaque bande

à l'intérieur de la couche avec comme approche un KP. Nous aurions pu résoudre avec un KP une étagère complète, mais cela n'est vraiment pas raisonnable en terme de temps d'exécution. Donc nous choisissons de n'appliquer les KP qu'aux cases (rangées). Après nous pouvons aussi faire comme A. Lodi et al. [2] en faisant une collection d'hauteurs d'étagères et résoudre un problème de Bin Packing 1D.
Cette approche ressemble fortement à celle de Pisinger. En effet, après l'initialisation d'une couche Pisinger résoud de façon optimale chaque bande. La différence est la façon d'initialiser une couche. Pisinger sélectionne plusieurs boîtes puis pour chacune d'entres elles, remplie la couche en faisant un KP sur chaque bande de la couche et récupère seulement le meilleur. Le second point qui diffère, est le fait que Pisinger permet de faire des couches horizontales ou verticales selon le meilleur résultat, exemple voir 4.1. Nous ne voulons pas pouvoir faire ça dans notre cas, car seules des couches horizontales sont permises.

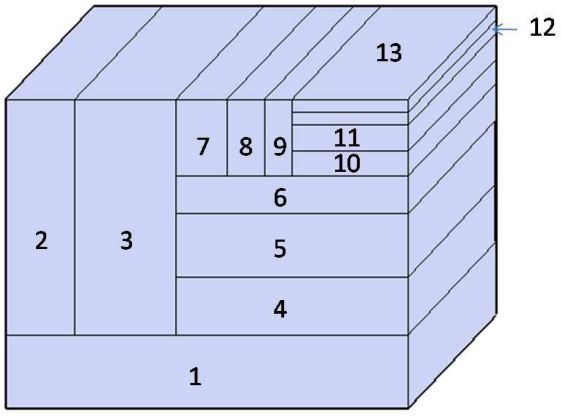

Figure 4.1: Différentes couches possibles avec Pisinger

4.7 Evaluation

Dans cette section nous présentons des résultats d'exécution de 3 algorithmes différents (dont 2 possédant chacun deux variantes). Il s'agit d'algorithmes que l'on a adaptés pour le bin packing 3D.
Ces algorithmes sont comparés en termes de performances et de temps d'exécution sur des données ayant des caractéristiques très proche de la réalité.
Le système avec lequel ont été effectués les tests est un intel(R) Core(TM)2 CPU 2.13GHz, avec 2.0Go de mémoire.
Nous prenons comme taille de conteneurs pour les tests [730,950,580], la taille des conteneurs a été choisie par rapport aux données industrielles réelles.

4.7.1 Test sur le taux de remplissage

Pour les dimensions des boîtes, 3 valeurs sont tirés aléatoirement entre 4 et 200 (également dû aux données réelles), puis les chiffres sont triés de manière à obtenir hauteur<largeur<longueur.
Dans le tableau 4.1 nous présentons pour les 5 heuristiques testées sur nos instances aléatoires.
La colonne remplissage correspond au taux de remplissage du conteneur.
La colonne temps correspond à au temps d'exécution de l'algorithme.
La colonne nb bins correspond au nombre de conteneurs utilisé par les boîtes.

	Algorithme	remplissage	temps(en s)	nb bin
10-50	NFDH	0,59632	0	4
boîte par	FFDH	0,88856	0,07	3
type	FFDHBIS	0,88947	0,07	3
(100 types),	BFDH	0,88879	0,11	3
2496 boîtes	BFDHBIS	0,89046	0,07	3
50-100	NFDH	0,72054	0,01	10
boîte par	FFDH	0,855517	0,11	9
type	FFDHBIS	0,856901	0,12	9
(100 types)	BFDH	0,862585	0,21	9
5475 boîtes	BFDHBIS	0,86068	0,15	9
100-150	NFDH	0,684491	0,02	20
boîte par	FFDH	0,976912	0,49	17
type	FFDHBIS	0,977392	0,44	17
(100 types)	BFDH	0,974611	0,81	17
11890 boîtes	BFDHBIS	0,974804	0,63	17
150-200	NFDH	0,733014	0,04	22
boîte par	FFDH	0,93583	1,01	19
type	FFDHBIS	0,936159	0,75	18
(100 types)	BFDH	0,934316	1,32	18
1942 boîtes	BFDHBIS	0,934556	0,97	18
200-250	NFDH	0,674317	0,03	38
boîte par	FFDH	0,95078	1,21	19
type	FFDHBIS	0,96041	0,8	18
(100 types)	BFDH	0,966208	1,78	18
25545 boîtes	BFDHBIS	0,962612	1,12	18

Table 4.1: Evaluation des algorithmes de packing

Il n'y a pas de différence notable dans le Tableau 4.1 entre les algorithme FFDH et BFDH en terme de remplissage. Cela semble indiquer que les instances tirées de cas réel ne sont pas réellement difficiles.

Les tests du tableau 4.2 sont effectués pour pallier aux instances trop faciles. Toutes les boîtes du tableau 4.2 sont différentes et n'ont pas la même taille, celle-ci est tirée entre

4 et 400.

	Algorithme	remplissage	temps(en s)	nb bin
5000 boîtes par type	NFDH	0,504416	0,01	78
	FFDH	0,945936	0,13	51
	FFDHBIS	0,94531	0,12	51
	BFDH	0,963684	0,2	50
	BFDHBIS	0,964275	0,14	50
5000 boîtes par type	NFDH	0,504416	0	73
	FFDH	0,939355	0,11	49
	FFDHBIS	0,939318	0,14	49
	BFDH	0,959074	0,19	48
	BFDHBIS	0,959047	0,15	48
10000 boîte par type	NFDH	0,503922	0,03	148
	FFDH	0,957723	0,52	96
	FFDHBIS	0,955989	0,44	96
	BFDH	0,966274	0,73	95
	BFDHBIS	0,966714	0,54	95
10000 boîte par type	NFDH	0,499965	0,02	146
	FFDH	0,95679	0,53	97
	FFDHBIS	0,955098	0,42	97
	BFDH	0,965783	0,72	96
	BFDHBIS	0,96604	0,52	96
100000 boîte par type	NFDH	0,499509	0,19	1489
	FFDH	0,974373	98,56	937
	FFDHBIS	0,9775679	44,81	936
	BFDH	0,986001	117,4	935
	BFDHBIS	0,980731	52,39	935
100000 boîte par type	NFDH	0,502406	0,19	1489
	FFDH	0,0,974397	98,26	937
	FFDHBIS	0,974415	44,01	936
	BFDH	0,976624	116,9	935
	BFDHBIS	0,976956	51,53	935
200000 boîte par type	NFDH	0,50025	0,38	2972
	FFDH	0,0,975862	415,75	1867
	FFDHBIS	0,976243	190,73	1866
	BFDH	0,9782	482,74	1864
	BFDHBIS	0,978508	219,46	1864
200000 boîte par type	NFDH	0,500580	0,38	2972
	FFDH	0,97632	413,63	1835
	FFDHBIS	0,976522	189,08	1833
	BFDH	0,978949	481,57	1833
	BFDHBIS	0,978959	214,9	1833

Table 4.2: Evaluation des algorithmes de packing avec des boîtes toutes différentes

Dans les résultats du tableau 4.2 nous voyons que la différence du taux de remplissage entre les algorithmes FFDH et BFDH se creuse, cela est visible également sur le nombre

de conteneurs après l'exécution des algorithmes.
Les performances des algorithmes étant fortement dépendantes des caractéristiques des boîtes, il me semble particulièrement pertinent d'inclure lors d'une extension future de ce travail une étude probabiliste et statistique sur les données. Ainsi une telle étude permettra éventuellement de choisir systèmatiquement l'algorithme le mieux adapté aux différentes instances.

4.7.2 Temps d'exécution des différents algorithmes

On peut facilement constater que les temps d'éxécution dans les tableau 4.1 et 4.2 sont très différents selon les algortihmes. L'algorithme NFDH est extrêmement rapide, même avec 200000 boîtes à ranger il ne met que 0,38 secondes, tandis que l'algorithme BFDH met 481-482 secondes. Si l'on compare le temps d'exécution entre FFDH et BFDH, on remarque que FFDH est plus rapide dans tous les cas. Pourtant FFDH et BFDH effectuent le même nombre d'opérations dans le pire des cas, soit n^4, mais l'algorithme BFDH exécute toujours en n^4 opérations, alors que pour que FFDH arrive a en réaliser n^4 il faudrait qu'une seule boîte puisse entrer par case. Cependant nous voyons que l'algorithme BFDHBIS donne de meilleurs résultats que l'algorithme FFDH. Cela s'explique par le fait que les algortihmes FFDHBIS et BFDHBIS dans le pire des cas réalise n^2 opérations.

4.7.3 Conclusion des évaluations

Les tests ont montré de grandes différences entre les algorithmes non seulement au niveau du temps d'exécution mais également en taux de remplissage des conteneurs. Deux algorithmes sur trois ont des résultats interessants notamment leur version améliorée, et pourraient être pris en compte selon les instances de boîtes que nous avons. En effet si les instances sont faciles et que le temps est notre principal critère, l'algorithme FFDH est une bonne solution. Par contre avec des instances un peu plus dures nous allons plutôt opter pour l'algorithme BFDH si le temps d'exécution n'est pas le critère premier.
Cependant pour le problème particulier qui nous a été posé par INTECUM, il s'agit de manipuler des instances de moins de 10000 boîtes. On peut constater que tout les algorithmes testés dans cette section s'exécuteraient en moins d'une minute pour de telles instances. Il s'agit là d'un temps très raisonnable car l'algorithme n'est déroulé qu'une seule fois par jour par le robot. Alors le choix de l'algorithme ne dépend finalement que des performances en terme d'espace utilisé.

Algorithm 2 FFDH

Require: List *boites* , List *conteneurs*

1: *conteneurs*.ajouteConteneur(), *conteneurCourant*=*conteneurs*[0]
2: *boites*.trierParHauteurDecroissante()
3: **for** $i = 0$ to *boites*.size() **do**
4: *pasDePlacementPossible*=true
5: $boiteCourante = boites[i]$
6: i=0, $j = 0$,k=0 {on va parcourir les cases, pour placer la boite dans le meilleur emplacement possible}
7: **while** (*pasDePlacementPossible*) and (i< *conteneurs*.size()) **do**
8: **while** (*pasDePlacementPossible*) and (j< *conteneurs*[i].*etageres*.size()) **do**
9: **while** (*pasDePlacementPossible*) and (k< *conteneurs*[i].*etageres*[j].*cases*.size()) **do**
10: **if** *conteneurs*[i].*etageres*[j].*cases*[k].peutContenir(*boiteCourante*) **then**
11: *conteneurs*[i].*etageres*[j].*cases*[k].ajoute(*boiteCourante*), *pasDePlacementPossible*=false
12: **end if**
13: i++
14: **end while**
15: j++
16: **end while**
17: k++
18: **end while**
{Pas trouvé de place pour la boite courante, on crée une case et essayer de la placer dans une étagère existante}
19: **if** *pasDePlacementPossible* **then**
20: i=0, $j = 0$, *caseCourante*=creerCase(*boiteCourante*)
21: **while** (*pasDePlacementPossible*) and (i< *conteneurs*.size()) **do**
22: **while** (*pasDePlacementPossible*) and (j< *conteneurs*[i].*etageres*.size()) **do**
23: **if** *conteneurs*[i].*etageres*[j].peutContenir(*caseCourante*) **then**
24: *conteneurs*[i].*etageres*[j].ajoute(*caseCourante*), *pasDePlacementPossible*=false
25: **end if**
26: i++
27: **end while**
28: j++
29: **end while**
{Pas trouvé de place pour la case courante, on va créer une étagère et essayer de la placer dans un conteneur existant}
30: **if** *pasDePlacementPossible* **then**
31: i=0, *etagereCourante*= creerEtagere(*caseCourante*)
32: **while** (*pasDePlacementPossible*) and (i< *conteneurs*.size()) **do**
33: **if** *conteneurs*[i].peutContenir(*caseCourante*) **then**
34: *conteneurs*[i].ajoute(*etagereCourante*), *pasDePlacementPossible*=false
35: **end if**
36: i++,
37: **end while**
38: **if** *pasDePlacementPossible* **then**
39: *conteneurs*.ajoute(creerConteneur*etagereCourante*))
40: **end if**
41: **end if**
42: **end if**
43: **end for**

Algorithm 3 BFDH2

Require: List *boites* , List *conteneurs*, List *cases*, List *etageres*

```
1: conteneurCourant, etagereCourante, meilleurCase
2: conteneurs.ajouteConteneur(),conteneurCourant=conteneurs[0]
3: boites.trierParHauteurDecroissante()
4: for i = 0 to boites.size() do
5:    trouverCase=false
6:    for (j=0 to j< cases.size()) do
7:       if cases[j].peutContenir(boites[i]) and cases[j].meilleurQueCaseCourante() then
8:          meilleurCase=cases[j],trouveCase=true
9:       end if
10:   end for
11:   if trouveCase==true then
12:      meilleurCase.ajoute(boites[i])
13:   else
14:      cases.ajoute(creerCase(boites[i]))
15:   end if
16: end for
    {Nous avons maintenant une collection de cases à placer dans les conteneurs}
17: for i = 0 to cases.size() do
18:   trouverEtagere=false
19:   for (j=0 to j< etageres.size()) do
20:      if etageres[j].peutContenir(cases[i]) and etageres[j].meilleurQueEtagereCourante()
      then
21:         meilleurEtagere=etageres[j],trouveEtagere=true
22:      end if
23:   end for
24:   if trouveEtagere==true then
25:      meilleurEtagere.ajoute(cases[i])
26:   else
27:      etageres.ajoute(creerEtagere(cases[i]))
28:   end if
29: end for
    {Nous avons maintenant une collection d'etagere à placer dans les conteneurs}
30: for i = 0 to etageres.size() do
31:   trouverConteneur=false
32:   for (j=0 to j< conteneurs.size()) do
33:      if conteneurs[j].peutContenir(etageres[i])
      and etageres[j].meilleurQueConteneurCourant() then
34:         meilleurConteneur=conteneurs[j], trouverConteneur=true
35:      end if
36:   end for
37:   if trouverConteneur==true then
38:      meilleurConteneur.ajoute(etageres[i])
39:   else
40:      conteneurs.ajoute(creerConteneur(etageres[i]))
41:   end if
42: end for
```

Chapitre 5

Pilotage optimal du module, simulation

5.1 Introduction

Dans ce chapitre nous voulons traiter l'évolution (dynamique) des conteneurs au cours du temps. L'objectif est d'organiser le flux des boîtes dans les conteneurs (retrait et dépôt de boîtes) pour maximiser le profit total entré dans les armoires. Cela revient à résoudre un problème multi-sac à dos multi-période avec retrait d'objets.
Pour résoudre ce problème, nous supposons que nous connaissons dès le départ toutes les boîtes qui vont rentrer dans les conteneurs, ainsi que leur durée de vie et leur date d'arrivée dans la simulation.
Pour commencer nous devons initialiser les conteneurs en résolvant le problème de bin packing 3D. Cette initialisation ne permet pas seulement de remplir les conteneurs, mais également de structurer ceux-ci, c'est à dire de pouvoir placer les étagères et les cases des conteneurs. Les étagères ainsi que les cases sont placées définitivement dans le temps. Ainsi chaque case pourra être considérée comme un sac à dos, d'où l'aspect multi-sac. Pour l'initialisation plusieurs algorithmes peuvent être appliqués avec des performances diverses. Nous avons donc choisi utiliser l'algorithme BFDH pour ces performances montrées dans le chapitre précédent. Cependant d'autres algorihmes, avec des performances comparables, comme FFDH, peuvent également être utilisés.
La seconde étape de résolution (après initialisation) est réalisée avec plusieurs algorithmes différents. Les résultats obtenus pour chacun d'eux sont présentés, analysés et comparés à la fin de ce chapitre.

5.2 Basic Algorithm Without Forward Looking (BAWFL)

Cette approche est la version la plus simpliste possible, elle n'utilise pas la connaissance que nous avons sur les futures entrées de boîtes (dates d'arrivée et dates de sortie). L'objectif ici est d'obtenir des premiers résultats de performances auxquels pourraient être comparés ceux d'algorithmes plus élaborés développés plus loin dans ce chapitre.

Nous commencçons par effectuer l'initialisation. Puis au jour t de la simulation, nous retirons les boîtes qui doivent sortir de la simulation à cet instant. Et nous trions les boites qui doivent rentrer à l'instant t selon le critère ci dessous

- $boite_A$ avant $boite_B$
- ssi $\frac{Poid_A}{Volume_A * Duree_A} > \frac{Poid_B}{Volume_B * Duree_B}$

Nous parcourons la liste des boîtes ainsi triées en essayant de les placer une par une. Nous plaçons donc en premier la boîte la plus intéressante que l'on puisse ranger. Nous passons ensuite à l'instant t+1.
L'approche répète l'opération pour chaque jour de la simulation. Les étapes de cette méthode sont détaillées dans l'algorithme 4.

Algorithm 4 BAWFL

Require: List *Jours*, *problemeInitiale*

```
1: conteneurCourant, etagereCourante, caseCourante
2: problemeInitiale.resoudBDFH();
3: for t=0 to t<Jours.size() do
4:    if t==0 then
5:       Jours[t]=problemeInitiale
6:    else
7:       Jours[t]=Jours[t-1]
8:    end if
9:    for j=0 to Jours[t].nombreConteneurs do
10:      for k=0 to Jours[t].conteneurs[j].etageres.size() do
11:         for l=0 to Jours[t].conteneurs[j].etageres[k].cases.size() do
12:            for m=0 to Jours[t].conteneurs[j].etageres[k].cases[l].boites.size() do
13:               if Jours[t].conteneurs[j].etageres[k].cases[l].boites[m].duree==0 then
14:                  Jours[t].conteneurs[j].etageres[k].cases[l].supprimeBoite(m)
15:               end if
16:            end for
17:         end for
18:      end for
19:   end for
20:   Jours[t].trierBoiteAmettre()
21:   for k=0 to Jours[t].boitesAmettre.size() do
22:      Jours[t].placeSelonBF(boitesAmettre[k])
23:   end for
24: end for
```

5.3 Résolution avec connaissance des boîtes entrantes à t+1

Dans cette partie, nous proposons des algorithmes résolvant le problème, pas à pas, en prenant compte le pas suivant. C'est à dire, pour chaque étape t nous résolvons l'instance du problème en connaissant les boîtes entrantes à t+1. La question qu'il faut donc se poser est : 'Au temps t, j'ai mis des boîtes, que se serait-il passé au temps t+1 (quelle boîte aurais-je pu mettre) si je n'avais pas rangé ces boîtes?'.
Deux versions, basées sur ce principe, sont détaillées dans la suite de cette section.

5.3.1 One Step Forward Looking Strategy (OSFLS)

Pour cette première version à connaissance t+1, nous commençons tout d'abord par faire une initialisation comme dans la version précédente. On résout avec l'algorithme BDFH pour rentrer un maximum de boîtes dans les conteneurs.
Puis à l'instant t, nous résolvons les instances t et t+1 naïvement. Nous conservons les boîtes qui ne sont pas entrées à l'instant t+1 en les triant avec le critère suivant

- $boite_A$ avant $boite_B$
- ssi $\frac{Poid_A}{Volume_A * Duree_A} > \frac{Poid_B}{Volume_B * Duree_B}$

Nous rangeons ces boîtes dans une liste $boitesNonRangees$. Ainsi le premier élément de la liste $boitesNonRangees$ est l'élément le plus intéressant de l'instant t+1 qui n'a pas pu être placé.
Maintenant nous allons démarrer la deuxième phase de l'algorithme en utilisant les connaissances que nous avons. Nous allons parcourir les boîtes rentrées dans les conteneurs à l'instant t, et regarder si une boîte plus rentable, que celle actuellement positionnée, peut rentrer à l'instant t+1 si l'on ne rangeait pas celle-ci. Cet algorithme a été défini et implémenté au cours du stage.

5.3.2 One Step Forward Looking Strategy 2 (OSFLS2)

Cette approche est basée sur le même principe que l'algorithme OSFLS, nous voulons être plus efficace.
Au moment de la résolution à l'instant t, nous conservons la liste($boitePlacee$) des boites entrantes à t qui sont triées de la façon suivante

- $boite_A$ avant $boite_B$
- ssi $\frac{Poid_A}{Volume_A * Duree_A} < \frac{Poid_B}{Volume_B * Duree_B}$

Nous les trions dans le sens inverse de la liste $boitesNonRangees$. Ainsi le premier élément de la liste $boitesPlacees$ est la boîte la moins intéressante que nous avons placée à l'instant

Algorithm 5 OSFLS

```
Require: List Jours, problemeInitiale, List boiteNonRangees, boiteCourante
 1: problemeInitiale.resoudBDFH();
 2: for t=0 to t<Jours.size()-1 do
 3:    if t==0 then
 4:       Jours[t].resoudAvecApprocheNaïve()
 5:    end if
 6:    Jours[t+1].resoudAvecApprocheNaïve()
 7:    for j=0 to problemeInitiale.nombreConteneurs do
 8:       for k=0 to Jours[t].conteneur[j].etageres.size() do
 9:          for l=0 to Jours[t].conteneur[j].etageres[k].cases.size() do
10:             for m=0 to Jours[t].conteneur[j].etageres[k].cases[l].boites.size() do
11:                boitesCourante= Jours[i].conteneur[j].etageres[k].cases[l].boites[m]
12:                if boitesCourante.date=t then
13:                   n=0, trouve=false
14:                   while n<=boitesNonRangees.size() and trouve=false do
15:                      if boitesCourante.valeur()<boitesNonRangees[n].valeur() and
                         boitesCourante.peutContenir(boitesNonRangees[n]) then
16:                         Jours[t+1].conteneur[j].etageres[k].cases[l].boites[m]=boitesNonRangees[n]
17:                         Jours[t].conteneur[j].etageres[k].cases[l].supprime(boiteCourante)
18:                      end if
19:                      n++
20:                   end while
21:                end if
22:             end for
23:          end for
24:       end for
25:    end for
26: end for
```

t. En plus des boîtes placées nous sauvegardons un pointeur sur le conteneur où elles ont été placées.

De cette façon nous allons pouvoir remplacer directement les boites les moins intéressantes par les boites les plus intéressantes à l'instant t+1 qui sont quand à elles dans la liste *boitesNonRangees*.

L'algorithme 6 illustre les étapes de cette approche.

5.4 Résolution avec connaissance de toutes les boîtes entrantes au cours de la simulation

Nous souhaitons profiter de l'ensemble des informations que nous avons et résoudre l'instance t ayant comme information les boîtes entrantes à t_n, n $\in$ jours de simulation possibles.

Algorithm 6 OSFLS2

Require: List *Jours*, *problemeInitiale*, List *boiteNonRangees*, List *boitesPlacees*, *boiteCourante*

{Initialisation avec résolution du Bin Packing 3D du départ}

```
1: problemeInitiale.resoudBDFH();
2: for t=0 to t<Jours.size()-1 do
3:    if t==0 then
4:       Jours[t].resoudAvecApprocheNaïve()
5:    end if
6:    Jours[t+1].resoudAvecApprocheNaïve() {Nous avons dans la liste botesPlacees la
      liste les boîtes placee à t, ainsi que dans la liste boiteNonRangees les boites non
      rangées à t+1}
7:    for i=0 to boitesPlacees.size() do
8:       j=0, trouve=false
9:       while j<boiteNonRangees.size and trouve==false
         and boitesPlacees.valeur()<boitesNonRangees[j].valeur() do
10:         if boitesPlacees.peutContenir(boitesNonRangees[j]) then
11:            Jours[t+1].remplace(boitesPlacees,boitesNonRangees[n])
12:            Jours[t].supprime(boitesPlacees)
13:            trouve=true
14:         end if
15:         j++
16:      end while
17:   end for
18: end for
```

5.4.1 Multiple Step Forward Looking Strategy (MSFLS)

Après la phase d'initialisation, nous récoltons toutes les boîtes que nous voulons faire rentrer au cours de la simulation. Puis nous les trions selon le critère suivant

- $boite_A$ avant $boite_B$
- ssi $\frac{Poid_A}{Volume_A * Duree_A} > \frac{Poid_B}{Volume_B * Duree_B}$

Puis nous allons simplement parcourir cette liste en plaçant si possible les boîtes. Au lieu de placer les boîtes les plus intéressantes à l'instant t, nous plaçons les boîtes les plus intéressantes au cours de toute la simulation. Cela pourrait s'assimiler à de l'ordonnancement de tâches (où les dates de début et de fin seraient fixées).
Cette approche est illustrée dans l'algorithme 7.

5.4.2 Multiple Step Forward Looking Strategy 2 (MSFLS2)

Cette approche est une sorte d'hybridation les algorithme OSFLS2 et MSFLS.
En effet nous commençons par résoudre le problèmpe avec l'algorithme OSFLS2. Dans la seconde partie de l'algorithme nous récupérons, dans une liste, toutes les boîtes que non rangées au cours de la simulation. Nous trions celles-ci pour avoir comme premier

Algorithm 7 MSFLS

Require: List *Jours*, *problemeInitiale*, List *boiteNonRangees*, List *boitesPlacees*, *boiteCourante*

```
1: for t=0 to t<Jours.size() do
2:    boiteNonRangees.ajouteBoitesDuJour(t)
3: end for
4: boiteNonRangees.trier()
5: for t=0 to t<boiteNonRangees.size() do
6:    possible=false
7:    i=0 {dans meilleurEmplacementPossible, nous avons la liste des emplacements
      possible au temps boiteNonRangees.date, ceux-ci sont triés du meilleur au moins
      bon}
8:    List meilleurEmplacementPossible= calculeLesEmplacementsPossible()
      {Nous devons vérifier si le meilleurEmplacementPossible est également libre tout
      au long de la duree de vie de notre boîte, sinon nous ne pouvons pas la placer}
9:    while possible==false and i<meilleurEmplacementPossible.size() do
10:      if possibleDePlacee(boiteNonRangees,meilleurEmplacementPossible[i]) then
11:         Jours[boiteNonRangees.date].placer(boiteNonRangees,meilleurEmplacementPossible[i
12:         possible=true
13:      end if
14:      i++
15:   end while
16: end for
```

élément: la meilleure boîte non placée.
Avec cette liste nous remplaçons les boîtes les moins intéressantes, que nous avons placées, par celle de la liste.

Nous allons donc faire des permutations. Ainsi cette méthode, qui est de la recherche locale, donnera forcément de meilleurs résultats que l'algorithme OSFLS2 (dans le pire des cas les mêmes résultats). En effet, vu que nous appliquons d'abord l'algorithme OSFLS2, et que nous faisons des permutations que si celles-ci nous apportent plus de gains, nous ne pouvons que gagner.
Cette version n'a pas été implémentée faute de temps

5.5 Réoptimisation intermédiaire

Dans les versions que nous avons présentées plus haut, nous n'avons pas pris en compte le fait de pouvoir réoptimiser à chaque jour de la simulation. En effet à partir du moment où des boîtes sortent du conteneur, un désordre dû aux trous créés pourrait apparaître. Ainsi une réorganisation, des boîtes restantes, pourrait être envisagée en vue de réoptimiser le placement.

5.6 Evaluation

5.6.1 Introduction

Les algorithmes étudiés (et/ou proposés) dans ce manuscrit ont été implémentés pour tester et comparer leur performances ainsi que leur temps d'exécution. Le système avec lequel ont été effectués les tests est un intel(R) Core(TM)2 CPU 2.13GHz, avec 2.0Go de mémoire.

La durée maximum d'une boîte passée dans le conteneur est tirée au hasard.
La taille des conteneurs est fixée à 730 en hauteur, 950 en largeur et 580 en profondeur.
Pour les dimensions des boites, 3 chiffres au hasard sont tirés entre 4 et 200, puis triés de manière à avoir hauteur<largeur<longueur.

5.6.2 Evaluation sur le temps des boîtes passées dans le conteneur

Dans cette sous-section nous évaluons l'influence que la durée de stockage des boîtes dans le conteneur et le nombre de jours de simulation peuvent avoir sur nos différents prototypes.
Le poids des boîtes est non corrélée a son volume et le nombre de type de boîtes différentes est de 50 pour le tableau 5.1 et 100 pour le tableau 5.4.
Le nombre de boîtes par type est compris entre 50 et 100.

Dans le tableau 5.1 nous présentons pour les 4 heuristiques testées sur nos instances aléatoires.
La colonne Poid tot. représente le profit total.
La colonne nb bins correspond au nombre de conteneurs utilisé par les boîtes.
La colonne remplissage correspond au taux de remplissage du conteneur.
La colonne durée max Boîte, correspond à au temps maximum que les boîtes peuvent rester dans le conteneur.

Dans le tableau 5.1, nous faisons varier la durée de présence des boîtes dans les conteneurs.

Nous voyons clairement que l'approche MSFLS(Algorithme 7) est beaucoup moins performant que les autres algorithmes. Le réel problème de cet algorithme est qu'il fragmente le remplissage de nos sacs. Pour comprendre le phénomène prenons un exemple simple: un seul sac ne pouvant contenir qu'une seule boîte à la fois et 5 boîtes pouvant toutes entrer dans le sac, nous supposons que les boîtes ont le même volume.
Nous avons 3 boîtes de type A, dont la valeur est 20.
-$boite_{A1}$ qui doit rentrer au temps T_0 et sortir au temps T_3
-$boite_{A2}$ qui doit rentrer au temps T_3 et sortir au temps T_6

	version	Poid tot.	nb bins	remplissage	durée max Boîte
1	BAWFL	24416647	6	0.754756	10
20 jours	OSFLS	24548164	6	0.745828	10
de simulations	OSFLS2	24944453	6	0.747299	10
(2326 boîtes)	MSFL	21530089	6	0.675932	10
2	BAWFL	21043160	4	0.89568	10
20 jours	OSFLS	2205557	4	0.88568	10
de simulations	OSFLS2	2205557	4	0.88568	10
(2305 boîtes)	MSFLS	18797126	4	0.820633	10
3	BAWFL	35387508	5	0.73568	10
20 jours	OSFLS	35372693	5	0.729136	10
de simulations	OSFLS2	35392329	5	0.734884	10
(2719 boîtes)	MSFLS	32472758	4	0.662181	10
4	BAWFL	39452279	4	0.86155	3
20 jours	OSFLS	39842790	4	0.8494	3
de simulations	OSFLS2	41034567	4	0.8543	3
(2297 boîtes)	MSFLS	39936285	4	0.78297	3
5	BAWFL	41099991	4	0.785434	3
20 jours	OSFLS	42558385	4	0.780569	3
de simulations	OSFLS2	44690322	4	0.787569	3
(2304 boîtes)	MSFLS	41702361	4	0.711257	3
6	BAWFL	43816649	4	0.795434	3
20 jours	OSFLS	44846798	4	0.787673	3
de simulations	OSFLS2	46454068	4	0.79087	3
(2313 boîtes)	MSFLS	42737353	4	0.711257	3

Table 5.1: Variation de la durée des boîtes

-$boite_{A3}$ qui doit rentrer au temps T_6 et sortir au temps T_9
Nous avons également 2 boîtes de type B, dont la valeur est de 25.
-$boite_{B1}$ qui doit rentrer au temps T_2 et sortir au temps T_5
-$boite_{B2}$ qui doit rentrer au temps T_7 et sortir au temps T_{10}
L'algorithme va donc classer nos boîtes selon le critère

- $boite_A$ avant $boite_B$
- ssi $\frac{Poid_A}{Volume_A * Duree_A} > \frac{Poid_B}{Volume_B * Duree_B}$

Nous allons donc avoir comme liste:
$boite_{B2}$,$boite_{B1}$,$boite_{A3}$,$boite_{A2}$, $boite_{A1}$
Nous allons placer les 2 premières boîtes, puis nous ne pourrons placer aucune des autres boîtes, dans notre conteneur nous aurons donc placé 2*25 points soit 50. Alors qu'avec nos autres algorithmes nous aurions obtenu 60 (Voir Figure 5.1).
Notre algorithme crée en effet des vides dans les sacs au cours du temps. Nous pouvons également le voir par les taux de remplissage de nos conteneurs qui est nettement plus bas dans la MSFLS par rapport à tous les autres algorithmes.

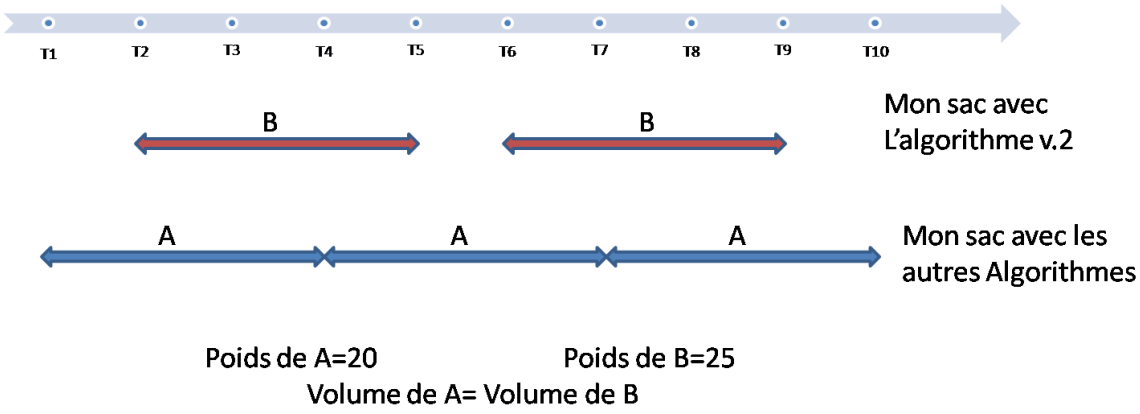

Figure 5.1: Exemple de déroulement d'un sac au cours de temps

Nous pouvons remarquer également que dans le tableau 5.1 que la simulation 3 donne un résultat légérement meilleur avec l'algorithme le plus naïf (35387508 contre 35372693) que l'algorithme OSFLS. On peut l'expliquer sur le faite que la simulation ne dure que 20 jours, et donc les algorithmes peuvent faire des choix de boîtes qui seront intéressantes quand elles sortiront un peu plus tard que les 20 jours de la simulation. Mais si l'on considérait une simulation avec un temps infini nous n'obtiendrons pas ce type de résultat (plus la simulation est longue, et moins cela apparaîtra).

Dans cette évaluation nous voulons connaître le comportement du prototype quand de nombreuses boîtes changent chaque jours. Il y a une différence remarquable entre les résultats selon le temps que les boîtes passent dans les sacs. En effet nous voyons que les différences des résultats entre l'approche naïve(BAWFL) et les approches de l'algorithme OSFLS et de l'algorithme OSFLS2, sont beaucoup plus nettes quand les boîtes ont une plus courte durée dans les conteneurs. De changer les objets plus fréquemment permet de remplacer très vite par une meilleure boîte, notamment lors de l'initialisation nous plaçons toutes nos boîtes sans regarder leurs valeurs.

5.6.3 Comparaison avec variation de la durée de la simulation

Dans le tableau 5.4 nous faisons la même chose que le tableau 5.1, mais ce que nous allons modifier est le nombre de jours de simulation (nombre de boîtes par type (1 à 50)).
Quand nous passons de 20 à 100 jours de simulation, la différence entre les résultats du dernier prototype (MSFLS) et les autres se creuse. Cela confirme bien les vides qui se forment dans les conteneurs avec la méthode MSFLS.

	version	Poids tot.	nb bin	remplissage
1	BAWFL	11198564	3	0.7549
20 jours	OSFLS	11089966	3	0.744189
de simulations	OSFLS2	11089966	3	0.744189
(2367 au départ)	MSFLS	10318369	3	0.714698
2	BAWFL	10151945	3	0.756252
20 jours	OSFLS	10151945	3	0.756252
de simulations	OSFLS2	10200519	3	0.760947
(1380 au départ)	MSFLS	9707470	3	0.719069
3	BAWFL	58757920	4	0.819051
100 jours	OSFLS	58844003	4	0.820535
de simulations	OSFLS2	59604553	4	0.829051
(2366 au départ)	MSFLS	47440036	4	0.794607
4	BAWFL	60576560	4	0.819051
100 jours	OSFLS	61732786	4	0.820535
de simulations	OSFLS2	61732786	4	0.829051
(2273 au départ)	MSFLS	48634607	4	0.774607

Table 5.2: variation du nombre de jours de simulation

5.6.4 Evaluation sur l'influence du poids

Nombre boîtes

- Entre 10 et 50 par type
- 100 types différents

D'après le tableau 5.3, nous remarquons que quand le poids des boîtes est fortement corrélé au volume de la boîte, nous obtenons beaucoup moins de différences entre les algorithmes, alors que quand le poids n'est pas corrélé, nous obtenons plus de différences entre les algorithmes.
Avec une corrélation à + ou - 10 pourcents sur le volume, les résultats des deux premières version sont sensiblement différents voir identiques. Alors que quand nous n'avons pas du tout de corrélation, les résultats sont nettement différents entre les algorithmes et l'algorithme OSFLS2 est clairement supérieure aux autres. David Pisinger a fait une étude sur les différences entre les instances dans les problèmes de sac à dos [11], dans son article il montre que les instances avec des objets dont le poids est corrélé au volume sont plus difficiles à résoudre que si elles sont non corrélées. Ce qui explique bien nos résultats. Si les instances sont entièrement corrélées, cela revient à résoudre un problème de bin packing classique, d'ailleurs nous avons présenté dans notre état de l'art sur le bin packing une version KP01.

	version	Poid tot.	nb bin
1	BAWFL	17671718	3
corrélé 10 pourcents	OSFLS	17742682	3
du volume	OSFLS2	17742682	3
(2332 boîtes)	MSFLS	17547572	3
2	BAWFL	27524722	4
corrélé 10 pourcents	OSFLS	27524722	4
du volume	OSFLS2	27524722	4
(2313 boîtes)	MSFLS	27142209	4
3	BAWFL	16175982	3
corrélé 10 pourcents	OSFLS	16175982	3
du volume	OSFLS2	16175982	3
(2323 boîtes)	MSFLS	16085766	3
4	BAWFL	21772041	4
corrélé 20 pourcents	OSFLS	21772041	4
du volume	OSFLS2	23535705	4
(2316 boîtes)	MSFLS	21608840	4
5	BAWFL	21178976	4
corrélé 20 pourcents	OSFLS	21969146	4
du volume	OSFLS2	21969146	4
(2335 boîtes)	MSFLS	20581164	4
6	BAWFL	14367999	3
corrélé 20 pourcents	OSFLS	14707112	3
du volume	OSFLS2	14707112	3
(2325 boîtes)	MSFLS	12804484	3
7	BAWFL	15057775	3
non corrélé	OSFLS	15396271	3
(2324 boîtes)	OSFLS2	15396271	3
	MSFLS	14305794	3
8	BAWFL	18154484	4
non corrélé	OSFLS	18161358	4
(2393 boîtes)	OSFLS2	18161358	4
	MSFLS	16750324	4
9	BAWFL	12026187	4
non corrélé	OSFLS2	12311529	4
(2333 boîtes)	OSFLS2	12311529	4
	MSFLS	11520162	4

Table 5.3: Résultat avec des boîtes corrélées (20 jours de simulations)

5.6.5 Comparaison du temps d'exécution des différents prototypes

Nous utilisons toujours la même taille de conteneur pour cette évaluation.
Nous utilisons ici 100 types de boîtes différents. La colonne temps(en s), représente le temps d'exécution des différents algorithmes.

Les résultats confirment bien que l'algorithme de l'algorithme OSFLS2 est plus rapide

	version	Poid total	nb bin	temps (en s)
1	BAWFL	60610881	3	0,1
10 jours	OSFLS	6135173	3	0,74
de simulations	OSFLS2	6142954	3	0,6
(1303 boîtes)	MSFLS	5755962	3	0,82
2	BAWFL	7272436	3	0,23
10 jours	OSFLS	7272436	3	0,38
de simulations	OSFLS2	7272436	3	0,31
(2288 boîtes)	MSFLS	7209156	3	0,88
3	BAWFL	24416647	6	0.38
20 jours	OSFLS	24548164	6	0.47
de simulations	OSFLS2	24944453	6	0,44
(2326 boîtes)	MSFLS	21530089	6	1,04

Table 5.4: Comparaison du temps d'exécution

que OSFLS, la complexité d'exécution de OSFLS2 étant inférieure à OSFLS. Tandis que l'algorithme MSFLS est plus lent que les autres, cela s'explique par le fait que quand nous avons trouvé une place libre pour une boîte b au temps t, il faut vérifier si cette place est libre durant toute la durée, au temps $t+1, t+2, ...t+n$, n étant la durée de vie de la boîte b.
Par conséquent l'algorithme MSFLS est complétement à proscrire puisque celui-ci est plus lent et moins efficace que les autres.

5.6.6 Conclusion des évaluations

Les différents tests que nous avons effectué montrent que les algoithmes OSFLS et OSFLS2 sont bien supérieurs aux autres méthodes développées. L'algorithme MSFLS est une grande déception, intuitivement ils nous semblait être meilleur. Mais l'algorithme MSFLS doit tout de même être testé avec la réoptimisation intermédiaire, comme pour tous les autres algorithmes. L'algortihme MSFLS2, qui n'a pas pu être testé, ne peut être que meilleur que les versions OSFLS et OSFLS2, tout en étant forcément moins rapide.

Chapitre 6

Conclusion et perspectives

Conclusion

Le problème traité dans ce manuscrit est un problème qui nous avait été soumis par le fabricant de robots INTECUM. Le but est de proposer des méthodes efficaces pour optimiser le rangement des boîtes de médicaments de façon systématique par le robot. La grande complexité de ce problème et l'absence totale de méthodes permettant sa résolution, dans la littérature, nous ont dissuadé d'envisager une éventuelle construction de nouvelles solutions à partir de zéro (from scratch).
Pour répondre à ce problème, nous l'avons donc découpé en deux phases, chacune correspondant à un sous-problème largement étudié dans la littérature. Une première phase pour structurer l'armoire, où nous avons modélisé une partie du problème comme un problème de bin packing 3D.
Dans un deuxième temps nous avons traité le pilotage de ce robot. Pour ceci, nous avons assimilé notre problème à celui du multi-sacs à dos multi-périodes avec retrait.
Dans ce mémoire nous avons donné un aperçu de certains travaux aboutis, de la littérature, sur les problèmes d'emballage en 2D et en 3D. Puis nous avons adapté certains algorithmes en 2D pour le 3D. Les résultats montrent que l'algorithme BFDH est un peu plus performant, au niveau remplissage de conteneurs, que l'algorithme FFDH, cependant plus lent. Selon les différentes instances, il serait donc plus intéressant d'utiliser l'algorithme FFDH.
En ce qui concerne le problème du pilotage du robot, nous avons parcouru la littérature des différentes variantes des problèmes de sac à dos et nous n'avons pas trouvé une variante semblable à la notre. Nous faisons donc plusieurs propositions pour résoudre ce problème.
Plusieurs des algorithmes étudiés dans ce travail, qu'ils soient de la littérature ou proposés durant ce stage, ont été implémentés et testés sur plusieurs instances de problèmes. Cet effort d'implémentation représente environ 4 000 lignes de code C++.

Perspectives

Pespectives à court terme

Tout d'abord, nous n'avons pas eu le temps d'implémenter et d'analyser les résultats que pourrait donner l'algorithme MSFLS2. Nous savons déjà que l'approche ne peut donner que de meilleurs résultats que la version OSFLS2 en terme de gains, puisqu'elle commence par dérouler l'algorithme OSFLS2 puis fait une recherche locale par dessus. Il serait donc intéressant d'implémenter cette méthode pour mesurer son apport de façon exacte et expérimentale.
La seconde perspective importante et de pouvoir revenir sur la structure de nos armoires de la première phase après avoir déroulé plusieurs fois le pilotage du robot. L'objectif étant d'utiliser les informations du mécanisme complet pour structurer les armoires. Une autre voie possible et souhaitable est l'utilisation de métaheuristiques.

Perspectives à long terme

Tout au long de ce travail, les aspects aléatoires n'ont pas été pris en considération. Je pense qu'il serait intéressant de reconsidérer le même problème dans un contexte encore plus général. C'est à dire avec des dates d'arrivées et de sorties aléatoires pour les boites du flux (en termes de nombre de boites) inconnu de façon certaine à priori. Le problème ainsi obtenu serait d'une grande complexité et hétérogène dans la nature de ses paramètres (certains stochastiques et d'autres pas). Il faudra alors explorer des approches stochastiques comme les processus aléatoires ou hybrides comme la simulation pour sa résolution.

Références

[1] A. Bortfeldt, et D. Mack , *A heuristic for the three-dimensional strip packing problem.,* European Journal of Operational Research, 183, 1267-1281 (2005).

[2] A. Lodi, S. Martello et D. Vigo , *Heuristic and Methaheuristic Approaches for a Class of Two-Dimensional Bin packing problem.,* INFORMS Journal on computing 11, pp 345-357 (1999).

[3] C. Kenyon et E. Remila , *A Near Optimal Solution to a Two-dimensional Cutting Stock Problem,* Mathematics of Operations Research, 25(2000), 645-656.

[4] D. Pisinger, *Heuristics for the container loading problem,* University of Copenhagen, Denmark, (2002).

[5] E.E. Bischoff et G. Wascher , *Cutting and Packing,* The European Journal of Operational Research, 84 (1995), 503-505.

[6] F.K.R. Chung, M.R. Garey et D.S. Johnson, *On packing two-dimentional bins,* SIAM J. algebraic Discrete Meth. 3(1982) 66-76.

[7] J.A. George et D.F. Robinson , *A heuristic for packing boxes into a container.,* Computers and Operations Research 7, 147-156 (1980).

[8] N. Ntene et J.H. Van Vuuren, *A survey and comparison of guillotine heuristics for 2D oriented offline strip packing problem,* Discrete Optimization (2009).

[9] P.C. Gilmore et R.E. Gomory, *Multistage cutting stock problems of two and more dimensions,* Operations Research 13 (1965) 94-120.

[10] S. Martello,M. Monaci et D.Vigo, *An Exact Approach to the Strip-Packing Problem,* INFORMS Journal on Computing, 15(3) (2003), 310-319.

[11] D. Pisinger, *Where are the hard knapsack?* Computers and Operations Research, 32:2271-2284 (2005).

[12] G.B. Mathews, *On the partition of numbers,* The London Mathematical Society, 28:486-490 (1897).

[13] F.Glover et G.A Kochenberger, *Critical event tabu search for multidimensional knapsack problems* I.H.OSMAN, J.P.KELLY, Eds., Metaheuristics : the Theory and Applications, p. 407-427, Kluwer Academic Publishers, Dordrecht, The Netherlands (1996).

[14] S.Hanafi et A.Fréville, *An approximate algorithm for multidimentional zeroone knapsack problems a parametric approach,* Management Science, vol. 43, p. 402-410 (1998).

[15] Y. Zhou et V. Naroditskiy, *Algorithm for stochastic multiple-choice knapsack problem and application to keywords bidding,* Proceeding of the 17th international conference on World Wide Web. New York, NY, USA: ACM, pp. 1175-1176 (2008).

[16] E.Y. Lin, *A Dynamic Programming Approach to the Multiple-Choice Multi-Period Knapsack Problem and the Recursive APL2 Code,* Presented at the 17th Triennial Conference of the International Federation of Operational Research Societies (2005).