République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Abderahmane MIRA -BEJAIA-

Faculté des Sciences Exactes

Département de Recherche Opérationnelle

Mémoire

Présenté par :

Mr TOUATI Sofiane

Pour l'obtention du diplôme de

Magister

Filière : Mathématiques Appliquées

Option : Modélisation Mathématique et Techniques de Décision

Thème

Résolution de Problèmes de Bin Packing à une dimension par la programmation DC

Devant le jury composé de :

Président	Mohand Ouamer Bibi	Professeur	Université de Béjaïa
Rapporteur	Mohammed Said Radjef	Professeur	Université de Béjaïa
Examinateur	Abdelkamel Tari	M.C.A	Université de Béjaïa
Examinateur	Mohamed Aidene	Professeur	Université de Tizi-Ouzou
Invité	Mohamed Maiza	Docteur	Ecole Militaire Polytechnique Bordj El Bahri

2013/2014.

Remerciements

***J**e tiens à remercier le professeur Mohammed Said RADJEF d'avoir accepté de m'encadrer, pour sa disponibilité, ses conseils et sa patience tout au long de ce travail.*

***J**e remercie le Professeur Mohand Ouamer BIBI d'avoir accepté de présider mon jury ainsi que pour l'attention accordée à mon travail.*

***J**e remercie tous les membres du jury de m'avoir fait l'honneur d'examiner mon travail.*

***J**e remercie aussi Mr Mohand BENTOBACHE pour ses conseils avisés, notamment pour l'implémentation informatique de mon travail.*

***M**es remerciements les plus vifs à ma famille pour leur encouragement.*

Dédicaces

Je dédie ce travail :

A *toute ma famille.*

A *mes amis.*

A *tous mes collègues de la post-graduation en mathématiques appliquées.*

Table des matières

Introduction générale

Les problèmes de bin packing ou plus généralement les problèmes de rangements et de découpes ont un intérêt certain tant du point de vue théorique que pratique. Tout d'abord, les problèmes réels liés aux problèmes de bin packing sont nombreux et variés [21; 65], on peut citer sans être exhaustif, les problèmes de découpe de matières premières, le placement optimal d'articles dans des journaux, enregistrement optimal de fichiers sur support mémoire, affectation de tâches à des processeurs, ...etc. Ces différents problèmes qui semblent éloignés les uns des autres ont en fait des caractéristiques communes, Dyckhoff [21] a élaboré une typologie pour ces problèmes en adoptant une notation semblable à la notation de Kendall pour les files d'attentes, Coffman et al. [18] en donne une autre plus récente intégrant les approches de résolution. Quand à l'aspect plus théorique, les problèmes de bin packing sont très étudiés par la communauté de la recherche opérationnelle et de l'optimisation combinatoire, ces problèmes connus pour être NP-complets [65] ont été traités de diverses manières : heuristiques et algorithmes d'approximations [4; 13; 16; 16; 20; 24; 28; 30; 33; 40], métaheuristiques [35; 36; 37; 47; 50; 56; 59; 61; 97], algorithmes exacts [14; 26; 34; 45; 67; 71; 85; 96]. Signalons aussi l'existence pour certaines versions du bin packing de liens avec l'ordonnancement (notamment sur machine parallèle) [9; 32; 70], et aussi un jeu basé sur le bin packing a été étudié par Kern et Oiu [41].

Dans ce travail, nous explorons une approche exacte par programmation DC, récemment Ndiaye et al. [71] ont appliqué une telle approche pour une version particulière du problème de bin packing à deux dimensions et Moeini et Thi [66] appliquent l'algorithme DCA à un problème de découpe en deux dimensions. La programmation DC [36; 52; 53; 92]

fait partie du domaine de l'optimisation globale, elle traite de la minimisation d'une différence de deux fonctions convexes, elle permet de modéliser sous forme d'un programme à variables continues tout programme linéaire à variables binaires (voir [79] pour l'approche continue pour les problèmes discrets), elle a été appliquée à plusieurs problèmes [2; 51; 54; 68; 72; 73; 75; 94]. L'algorithme obtenu par les conditions d'optimalité locale en programmation DC, appelé DCA est d'une forme très simple, l'algorithme procède par une linéarisation de la partie concave de la fonction objectif, le problème obtenu devient convexe, la résolution du programme DC revient alors à résoudre une succession de programmes convexes, la convergence de l'algorithme n'est que locale mais est en général meilleure que celles des algorithmes classiques.

Le présent mémoire est partagé en quatre chapitres. Dans le premier chapitre, nous donnons un bref aperçu de notion d'optimisation, complexité et d'algorithmes d'approximation.

Le second chapitre est consacré aux problèmes de bin packing à une et deux dimensions, notamment leurs modélisations et les méthodes de résolutions.

Dans le troisième, nous présentons la programmation DC et l'algorithme DCA, puis l'application de la programmation DC à la programmation linéaire à variables binaires et enfin un algorithme d'optimisation globale par séparation-évaluation, combiné à DCA.

Dans le quatrième chapitre, nous donnons les résultats numériques de l'application de la programmation DC, combinée à une procédure par séparation et évaluation aux problèmes de bin packing à une dimension avec et sans conflits.

Puis enfin une conclusion générale sur le travail effectué, et quelques perspectives.

Chapitre 1

Notions générales d'optimisation combinatoire

Dans ce chapitre, on présentera quelques éléments nécessaires à la compréhension du problème de bin packing de par sa nature (combinatoire) ainsi que sa complexité. On abordera très succinctement la problématique d'optimisation, quelques exemples, puis on donnera une classification simple des différentes classes de problèmes d'optimisation et on terminera par la complexité de la résolution de ces problèmes.

1.1 Notions d'optimisation

La problématique de l'optimisation apparaît souvent quand il s'agit de prendre une décision parmi plusieurs, elle est un outil fondamental dans la prise de décision. Divers domaines d'applications existent, comme la planification des vols d'une compagnie aérienne, la gestion des porte-feuilles financiers, la planification de la production, l'optimisation des réacteurs chimiques, ...etc. Une bibliographie très abondantes [46; 76; 89; 100] existe à ce sujet et traite des divers problèmes d'optimisation ainsi que des méthodes de résolution. Il s'agit en fait d'identifier un *objectif* pour mesurer la qualité d'un choix donné (gain financier, temps, consommation de ressource, ...), puis de determiner les éléments agissant sur cet objectif (*variables de décisions*), et la manière dont ils interagissent entre eux

(*contraintes du systèmes*), ce processus là est appelé *modélisation*. D'un point de vue mathématique, un problème d'optimisation est une minimisation ou bien une maximisation d'une certaine fonction (dite objectif) sous des contraintes. Notons par :

- x : le vecteur constitué des variables de décision, avec $x = (x_1, ..., x_n) \in \mathbb{R}^n$,
- f : la fonction objectif réelle,
- g : le vecteur des fonctions intervenant dans les contraintes.

Un problème d'optimisation peut alors être défini de manière générale de la manière suivante :

$$\begin{aligned} &\min f(x) \\ &\begin{cases} g_i(x) = 0 & i \in I_1, \\ g_i(x) \leq 0 & i \in I_2. \end{cases} \end{aligned} \tag{1.1}$$

Où I_1, I_2 sont les ensembles d'indices des contraintes de types égalités et inégalités respectivement. Soit $X = \{x \in \mathbb{R}^n : g_i(x) = 0, i \in I_1, g_i(x) \leq 0, i \in I_2\}$.

Définition 1.1.1. *Un vecteur $x \in \mathbb{R}^n$ est dit **solution réalisable** pour le problème (1.1) si x vérifie les contraintes du problème (ie, $x \in X$).*

Définition 1.1.2. ***(Minimum et maximum global)*** *Une solution réalisable x^* est dite minimum (resp. maximum) global de f sur X, si :*

$$f(x^*) \leq f(x) \quad (resp.\ f(x^*) \geq f(x)), \forall x \in X. \tag{1.2}$$

Définition 1.1.3. ***(Minimum et maximum local)*** *Une solution réalisable x^* est dite minimum (resp. maximum) local de f sur X, si :*

$$\exists \epsilon > 0 : f(x^*) \leq f(x) \quad (resp.\ f(x^*) \geq f(x)), \forall x \in X \bigcap B(x^*, \epsilon). \tag{1.3}$$

où : $B(x^*, \epsilon) = \{x \in \mathbb{R}^n, \|x^* - x\| \leq \epsilon\}$.

1.1.1 Exemples de problèmes d'optimisation

Exemple 1 : minimum et maximum d'une fonction

Dans la figure 1.1, on peut voir les optimum locaux et globaux de la fonction $f(x) = \frac{cos(3\pi x)}{x}$ sur [0,1.3].

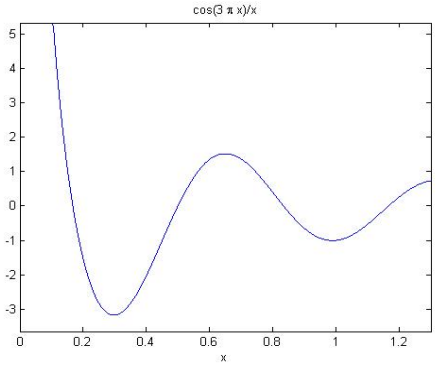

FIGURE 1.1: Représentation du graphe de la fonction $\frac{cos(3\pi x)}{x}$

Exemple 1 : dimension d'une citerne

Trouver les paramètres d'une citerne (forme cylindrique) tel que pour une surface extérieure S_0 donné, la citerne soit de contenance maximale.
Soit x_1 le diamètre de la base du cylindre et x_2 sa hauteur, il nous faut maximiser le volume donné par $f(x_1, x_2) = \pi(\frac{x_1}{2})^2 x_2$, et tel que : la surface totale donné par $g(x_1, x_2) = 2\pi(\frac{x_1}{2})^2 + \pi x_1 x_2$ soit égale à S_0, le diamètre et la hauteur sont strictements positifs. Le modèle s'écrit alors :

$$\begin{array}{l} \max f(x_1, x_2) = \pi(\frac{x_1}{2})^2 x_2 \\ \left\{ \begin{array}{l} 2\pi(\frac{x_1}{2})^2 + \pi x_1 x_2 = S_0, \\ x_1, x_2 \geq 0. \end{array} \right. \end{array} \tag{1.4}$$

Exemple 2 : problème du sac-à-dos

Considérons un ensemble d'objets indexés par $I = \{1, ..., n\}$, ayant chacun un poids w_i et une utilité u_i, et considérons un sac de contenance W. Le but est de ranger les objets sans excéder la capacité du sac, tout en maximisant l'utilité totale des objets rangés [66]. Le modèle s'écrit :

$$\begin{array}{l} \max \sum_{i=1}^{n} u_i x_i \\ \left\{ \begin{array}{l} \sum_{i=1}^{n} w_i x_i \leq W, \\ x_i \in \{0, 1\}, i \in I. \end{array} \right. \end{array} \tag{1.5}$$

où :

$$x_i = \begin{cases} 1, & \text{si l'objet i est rangé dans le sac ;} \\ 0, & \text{sinon.} \end{cases}$$

.

Exemple 3 : problème de la somme de sous-ensembles

Le problème de la somme de sous-ensembles (sub-set sum) [66] est un cas particulier du problème de sac à dos où les utilités u_i sont égales à w_i, et le but est de résoudre l'équation diophantienne $\sum_{i=1}^{n} w_i x_i = W$, on peut modéliser ce problème par le modèle suivant :

$$\begin{array}{l} \max \sum_{i=1}^{n} w_i x_i \\ \begin{cases} \sum_{i=1}^{n} w_i x_i \leq W, \\ x_i \in \{0,1\}, i \in I. \end{cases} \end{array} \tag{1.6}$$

Le problème a une solution si la valeur optimale de (1.6) est égale à W.

Exemple 4 : programme linéaire stochastique

Supposons que dans l'exemple 2, les poids ne sont pas connus de manière certaine mais sont des variables aléatoires, on peut par exemple chercher à maximiser l'espérance $E[\sum_{i=1}^{n} u_i x_i]$ [89].

1.1.2 Classification des problèmes d'optimisation

Les problèmes d'optimisation peuvent être classés suivant la nature de l'ensemble des solutions réalisables et la fonction objectif. Chacune des classes n'excluent pas forcément l'autre, de plus certains problèmes peuvent être reformulés et apparaître faisant partie aussi d'une autre classe. En voici une classification tirée de [76], récapitulé dans la figure 1.2.

Problèmes sans contraintes et problèmes contraints

Un problème est dit sans contraintes s'il est réduit à une minimisation de la fonction objectif sur son domaine de définition. On peut en fait réécrire tout problème avec

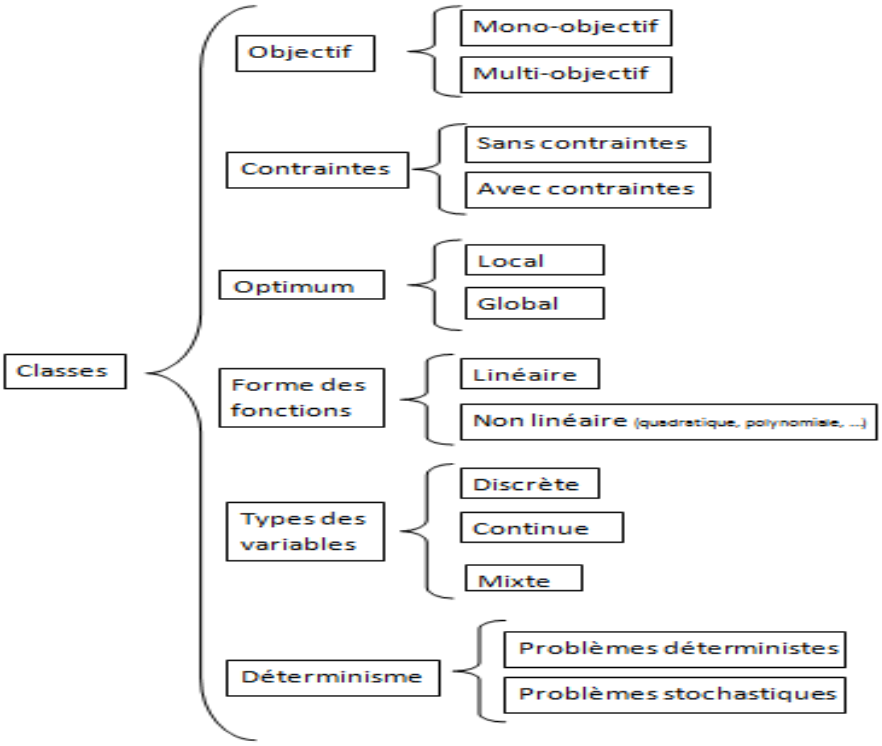

FIGURE 1.2: Classification des problèmes d'optimisations

contraintes en un problème sans contraintes (utilisation de pénalité, fonction barrière, fonction indicatrice), plusieurs algorithmes se basent sur une telle reformulation, le DCA (chapitre 3) en est un exemple.

Problèmes d'optimisation déterministes et problèmes stochastiques

Un problème d'optimisation est dit déterministe si chacun des éléments qui le caractérise n'est pas sujet à un aléa, contrairement aux problèmes stochastiques où l'élément aléatoire intervient. Par exemple, l'optimisation de la maintenance intégrera la nature aléatoire des pannes, par contre chercher le plus court chemin dans un graphe est un problème déterministe, étant donné qu'aucun élément aléatoire n'intervient.
Les problèmes stochastiques apparaissent souvent quand les paramètres du problèmes ne sont pas connus avec certitude mais fluctuent, et tel qu'on ne peut connaître à l'avance les paramètres exacts du problème.

Optimisation discrète et optimisation continue

Dans les problèmes de prise de décision, la modélisation du problème s'appuie parfois sur des variables binaires ou entières, par exemple le choix entre deux alternatives exclusives entraînera forcément des variables binaires. Plus généralement, quand les variables sont entières ($x \in \mathbb{Z}^n$), l'on parle alors de variables discrètes. Le problème d'optimisation fait appel à des techniques spécifiques aux problèmes d'optimisation discrète. En fait le nombre de solutions est souvent fini, mais il est exponentiel par rapport à la taille du problème (explosion combinatoire), et il s'avère beaucoup plus difficile en règle générale que l'optimisation à variables continues.

Optimisation locale et optimisation globale

Les algorithmes de résolution des problèmes d'optimisation s'appuie le plus souvent sur des conditions d'optimalités locale, au sens où la solution trouvée par de tels algorithmes est la meilleure dans un certain sous-ensemble des solutions possibles, dans certains cas particuliers (convexité), le minimum trouvé est global, de tels cas sont toutefois rares, et des approches nouvelles ont été explorées (métaheuristique, programmation DC) afin d'améliorer le plus possible la globalité des solutions trouvées. Sauf précision contraire, pour un problème d'optimisation, on entend trouver le minimum global de f et non pas le minimum local. Dans le cas non-convexe, la recherche d'optimums globaux est une tache difficile et exige des méthodes particulières pour traiter ces problèmes, on distingue alors optimisation locale et optimisation globale.

1.2 Notions de Complexité

La théorie de la complexité traite de la difficulté de résolution des problèmes formulés mathématiquement, cette difficulté est mesurée aussi bien en temps (temps nécessaire à sa résolution) qu'en espace (espace mémoire). Il est prévisible d'avoir une augmentation de ces quantités en fonction de la taille du problème, on s'intéresse alors à la manière dont elles augmentent. La pire situation étant que le temps nécessaire à la résolution d'un problème soit une fonction exponentielle de la taille du problème. La théorie de la complexité

étudie plus généralement les problèmes de décisions pour lesquels la réponse est soit "oui", soit "non".
En premier lieu, on doit distinguer entre un problème et une instance du problème. En effet on peut considérer un problème comme étant un ensemble d'instances de même structure. La théorie se base le plus souvent sur les problèmes de décision (un problème d'optimisation peut se ramener à un problème de décision relatif à l'existence d'une solution atteignant une certaine valeur objectif, en particulier l'optimum).

Définition 1.2.1. *[25] Un problème de décision est un ensemble I d'instances, partitionné en un ensemble I^+ d'instances positives (ayant la réponse oui), et I^- d'instances négatives (ayant pour réponse non). Résoudre le problème de décision consiste à déterminer pour une instance $i \in I$, si $i \in I^+$ ou si $i \in I^-$.*

La résolution d'un tel problème se fait par un algorithme, qui est en fait une succession d'opérations élémentaires effectuées par une machine, la notion d'algorithme a été plus formellement définie grâce aux machines de Turing, qui est un modèle théorique simplifié d'une machine.
Pour tout algorithme, on s'intéresse à quatre aspects fondamentaux, à savoir la finitude, la correction, la complétude et la complexité.

Définition 1.2.2. *Soit $\mathcal{A}$ un algorithme, on dit que :*

- *$\mathcal{A}$ se termine, s'il s'arrête en un temps fini.*
- *$\mathcal{A}$ est correct, si la réponse donné est oui pour une instance positive et non pour une instance négative.*
- *$\mathcal{A}$ est complet, s'il fournit toujours une réponse.*

L'aspect fondamental de la complexité d'un problème est le temps de résolution de ce problème. Deux grandes classes de problèmes notées **P** et **NP**, distinguent la complexité des problèmes de décision.

Définition 1.2.3. *[25] La classe **P** est l'ensemble des problèmes de décision pouvant être résolus en temps polynomial par une machine de turing déterministe.*

Définition 1.2.4. *[25] Un problème de décision appartient à* ***NP*** *s'il existe une relation binaire polynomiale* $\mathcal{R}$ *et un polynôme* p *tels que :*

$$\forall a \in I, a \in D^+ \Leftrightarrow \exists x : R(a,x) \text{ et } |x| \leq p(|a|) \tag{1.7}$$

La relation $\mathcal{R}$ est polynomiale si on peut determiner en temps polynomial en fonction de $|a| + |x|$ si $\mathcal{R}(a,x)$ ou non. Cette définition veut dire qu'un problème fait partie de la classe NP si pour toute instance positive de ce problème, l'on peut vérifier en temps polynomial que x est une réponse positive ou non.

Définition 1.2.5. ***(Réduction de Karp)****[25] Soient* $D_1 = \{D_1^+, D_1^-\}$ *et* $D_2 = \{D_2^+, D_2^-\}$ *deux problèmes de décision.* D_1 *se réduit à* D_2 *s'il existe une fonction* f *telle que :*

- $\forall a \in D_1, f(a) \in D_2$;
- $a \in D_1^+ \Leftrightarrow f(a) \in D_2^+$;
- f *est calculable en temps polynomial.*

Proposition 1.2.1. *[25] La réduction de Karp préserve l'appartenance à P, c'est-à-dire que si* $D_2 \in \boldsymbol{P}$ *et que* D_1 *se réduit à* D_2*, alors* $D_1 \in \boldsymbol{P}$*.*

Définition 1.2.6. *[25] Un problème* a $\in$ ***NP*** *est dit* ***NP****-complet, si tout problème de* ***NP*** *se réduit (réduction de Karp) à* a*.*

Celà montre que la classe **NP** est formée d'un noyau, tel que la résolution d'un seul de ses problèmes en temps polynomial entraîne la résolution de tout autre problème en temps polynomial.

1.2.1 Complexité temporelle des algorithmes

L'évaluation de la qualité d'un algorithme repose sur la qualité de la solution fournie ainsi que le temps mis à la trouver, la complexité temporelle concerne ce dernier critère et doit être définie de manière indépendante de la machine sur laquelle il est exécuté. On entend par temps, le nombre d'opérations élémentaires exécutées dans le pire cas jusqu'à ce que l'algorithme se termine. On utilise la notation en grand O pour exprimer ce nombre.

Notation	appellation
$O(1)$	constant
$O(\log(n))$	logarithmique
$O((\log(n)^c)$	polylogarithmique
$O(n)$	linéaire
$O(n\log(n))$	quasi-linéaire
$O(n^2)$	quadratique
$O(n^c), c \geq 1$	polynomiale
$O(c^n), c \geq 1$	exponentielle
$O(n!)$	factorielle

TABLE 1.1: Fonctions usuelles en complexité algorithmique [65]

Définition 1.2.7. *[65] Soient $f, g : \mathbb{R} \to \mathbb{R}$, on dit que f est dominée par g pour $x \to +\infty$, et on note :*

$$f(x) = O(g(x)) \text{ quand } x \to +\infty,$$

si $\exists x_0, C \in \mathbb{R}^+ : \quad \forall x > x_0 \quad |f(x)| \leq C|g(x)|$.

Définition 1.2.8. *[65] Soit A un algorithme ayant en entrée une donnée de taille n et $f : \mathbb{R} \to \mathbb{R}_+$. S'il existe une constante $\alpha > 0$ telle que A se termine après un nombre $\alpha f(n)$ opérations pour tout n, alors on dira que A s'exécute en O(f) (ou bien que la complexité temporelle de A est en $O(f)$).*

La complexité temporelle dépend donc de la fonction f, voici récapitulées dans la table 1, les plus utilisées :

Remarquons que la classe exponentielle $c^n, c \geq 1$ est totalement différente de la classe polynomial $O(n^c), c \geq 1$, à titre d'exemple : pour n=60, c=2 et si une opération élémentaire s'exécute en $1\mu s$, l'algorithme exponentiel se termine après 365.58 siècles alors que l'algorithme quadratique s'exécute en 0.0036s.

Remarque 1.2.1. *En pratique, pour les problèmes de la classe NP, les algorithmes exactes existants sont exponentiels. c'est à dire qu'il existe une instance pour laquelle l'algorithme nécessite un temps prohibitif, on ne s'intéresse alors qu'aux performances pratiques de tel*

algorithme, d'où l'intérêt de tests sur des problèmes réputés difficiles regroupés dans des benchmarks.

1.3 Méthodes de résolution

Les problèmes d'optimisations combinatoire, par le fait qu'ils sont bien souvent des problèmes NP-complet, sont traités par diverses manières, la première manière qui prévaut à toutes autres est d'essayer de trouver une solution réalisable à ce problème, puis on doit essayer d'améliorer la qualité de ces solutions. Il existe pour ce faire quatres types d'algorithmes pour la résolution de problèmes combinatoires. Nous en décrivons ici-bas leurs principales caractéristiques.

1.3.1 Heuristiques

Les heuristiques sont les algorithmes les plus simples et les moins coûteux à mettre en oeuvre pour tenter de résoudre un problème combinatoire NP-complet, le but premier d'une heuristique s'inscrit dans le souci de trouver une solution réalisable à un problème difficile à défaut de pouvoir le résoudre à l'optimum. L'inconvenient des heuristiques est donc qu'elles ne peuvent en règle générale fournir une solution optimale pour toute instance du problème, le problème posé au préalable reste donc entier, par contre les heuristiques ont plusieurs avantages, premièrement, ce sont des algorithmes polynomiaux et rapides, deuxièmement, ils se basent souvent sur la description du problème sans autres modélisations mathématiques et puis ils sont faciles à mettre en oeuvre. Le désavantage majeur des heuristiques ne diminue pas pour autant de leur intérêt, étant donné la complexité des problèmes NP-complets, ainsi dans toute démarche de résolution de problèmes combinatoires NP-complets, les heuristiques doivent être envisagées et utilisées avant toutes autres approches.

1.3.2 Métaheuristiques

Les métaheuristiques [78] sont des algorithmes d'optimisation visant à résoudre des problèmes NP-difficiles, ce sont en général des algorithmes intégrant une phase d'ap-

prentissage, inspirée de mécanismes naturels qu'ils soient physiques (algorithme du recuit simulé,...), biologique (algorithmes génétiques,...), ce sont des algorithmes plus généraux que les heuristiques au sens où ils ont été inventés dans le but de s'adapter facilement à une large gamme de problèmes. Il existe plusieurs métaheuristiques différentes, mais suivent une approche générale d'exploration de l'espace des solutions de manière diversifiée et répétée ainsi qu'une phase d'apprentissage afin de cibler la recherche de façon efficace et de manière à éviter de se faire piéger par des optimums locaux (ce qui est souvent le cas des algorithmes exacts), notons aussi que les heuristiques sont souvent ajoutées au schéma général d'une métaheuristique. On trouve parmi les métaheuristiques les plus connues, les algorithmes génétiques, le recuit simulé, les algorithmes d'optimisation par essaim particulaire, les algorithmes d'optimisation par colonie de fourmis. L'avantage de tel algorithme est qu'il peuvent être adapté d'un problème à un autre. Le désavantage majeures des métaheuristiques est qu'elles ne peuvent assurer l'optimalité de la solution fournie, mais donnent en pratique des solutions très souvent satisfaisantes.

1.3.3 Algorithmes exacts

Les algorithmes exacts sont un type d'algorithmes assurant l'optimalité de la solution. Ce sont dans le cas NP-complet des algorithmes nécessitant un nombre exponentiel d'itérations au pire des cas. On trouve deux grandes classes d'algorithmes exacts en optimisation combinatoire : les algorithmes de type séparation-évaluation (SE). Les algorithmes SE construisent des solutions partielles au problème puis tentent de les compléter. Les méthodes de coupe construisent une suite de solutions non optimales que l'algorithme exclut de l'ensemble des solutions recherchées jusqu'à trouver une solution optimale.

1.3.4 Hyperheuristiques

Les hyperheuristiques [8] sont une classe nouvelle d'algorithmes d'optimisation dédiés au problèmes difficiles, l'idée première des hyperheuristiques était d'avoir un ensemble de stratégies automatisées dans le choix et le paramètrage d'heuristiques (cette automatisation étant elle-même heuristique, les hyperheuristiques étaient pensées pour être des heuristiques pour **choisir** d'autre heuristiques), puis un nouveau type est apparu avec

l'intégration de la programmation génétique et n'est plus limité au choix mais a pour but de **générer** des heuristiques.

1.4 Notion d'algorithmes d'approximation

Considérons un problème d'optimisation combinatoire NP-complet (Π), pour lequel trouver un algorithme exact polynomial semble exclu. La première approche pourrait de trouver un algorithme exact satisfaisant en pratique (tout du moins sur les instances sur lesquelles il a été testé), une deuxième approche serait de considérer des algorithmes polynomiaux fournissant une solution réalisable aussi bonnes que possible, l'étude de tels algorithmes portera donc sur la qualité des solutions fournies (donc de l'algorithme) et la complexité de l'algorithme. Pour étudier la qualité des solutions, il nous faut une mesure de l'écart avec la solution optimale.
Notons par I une instance d'un problème de minimisation, noté Π, et OPT(I) la valeur minimale pour l'instance I, par A un algorithme polynomial et par A(I) la valeur de l'objectif pour la solution fournie par A.

Définition 1.4.1. *[46] Un algorithme d'approximation absolue pour* Π *est un algorithme polynomial A pour* Π *pour lequel il existe une constante k telle que :*

$$A(I) - OPT(I) \leq k \tag{1.8}$$

pour toutes instance I de Π*.*

Définition 1.4.2. *[46] Soit* $k \geq 1$*. Un algorithme d'approximation de facteur k pour* Π *est un algorithme polynomial A pour* Π *tel que :*

$$A(I) \leq kOPT(I) \tag{1.9}$$

Pour toutes instances I, et tel que : $A(I), OPT(I) \geq 0$*. On dit aussi que A a un rapport de performance (ou encore une garantie de performance) égale à k.*

Remarquons que si OPT(I) est nul, alors on doit nécessairement avoir une solution exacte pour que A soit un algorithme d'approximation, et pour $k = 1$, nous obtenons un algorithme polynomial exact.

Définition 1.4.3. *[25] Un algorithme d'approximation asymptotique de facteur k pour* Π *est un algorithme polynomial A pour lequel il existe une constante c telle que :*

$$A(I) \leq kOPT(I) + c \tag{1.10}$$

Pour toutes instances I de Π *et tel que* $OPT(I) \geq 0$*. On dit aussi que A a un rapport asymptotique égal à k.*

Chapitre 2

Des problèmes de bin packing

2.1 Caractéristiques fondamentales des problèmes de bin packing

2.1.1 Introduction

Les problèmes de bin packing existent sous différentes formes, chacune faisant référence aux problèmes pratiques traités, ainsi on rencontre des problèmes de :

- découpe et de réduction de perte (industrie du métal, bois),
- chargement de containers, palette (transport et logistique),
- allocation mémoire (informatique), ...etc

Ces quelques exemples illustrent l'étendue du champ d'application, et sous une apparente différence figure des similarités qui forment le problème de bin packing avec toutes ses variantes.

Dyckhoff [21] a été le premier à regrouper les différentes versions et en expliciter les différentes caractéristiques, que nous reprenons ici.

L'exemple type de problème de bin packing est celui de découpe de tube (décrit par Heicken et König en 1980), consistant à constituer des tubes de différentes longueurs et dans différentes quantités à partir d'un lot de tubes de longueurs suffisantes et cela en minimisant le nombre de tubes utilisés. Dans cet exemple, on remarque que la forme du tube

(cylindrique) ainsi que son diamètre ne compte pas dans le problème, en fait seul la longueur compte (problème à une seul dimension). Cet exemple de découpe peut néanmoins être vu comme un problème de rangement des tubes produits dans des cylindres d'une longueur finie, le but étant d'en utiliser le moins possible.
A partir de l'exemple précédent on peut imaginer d'autres extensions, comme des problèmes de découpe en deux dimensions ou de placement d'articles dans un journal (en placer un maximum dans une surface limitée), certain objets ne peuvent être placés côte à côte (conflit) ...

2.1.2 Caractéristiques

Un problème de bin packing est défini par les données d'une liste d'objets $\mathcal{O} = \{a_1, ..., a_n\}$ et d'une liste de bin (ou boite) $\mathcal{B} = \{b_1, ..., b_m\}$. Le but étant de ranger les objets dans les bins, c'est à dire de trouver une affectation (ou rangement) $\mathcal{P} : \mathcal{O} \to \mathcal{B}$, tout en minimisant un critère donné, en général le nombre de bins utilisés.
Cette définition assez générale englobe plusieurs variantes du problème de bin packing. Ce qui différencie un problème d'un autre repose sur les différentes caractéristiques des objets, bins ou de l'affectation.

Dimension

La dimension des objets et des bins est la caractéristique la plus importante dans les problèmes de bin packing.
On distingue trois grandes classes de BPP :

1. 1D-BPP problème de bin packing à une dimension,
2. 2D-BPP problème de bin packing à deux dimensions,
3. 3D-BPP problème de bin packing à trois dimensions.

Taille et orientation

La forme des objets et des bins dans les problèmes de bin packing est rectangulaire.
Par rapport à la taille des bins, on distingue deux types de variantes :

1. Les bins ont tous les mêmes mensurations, c'est le problème classique de BPP,
2. Les bins ont des mensurations différentes (il peut s'ajouter à cela un coût d'utilisation différent suivant la contenance du bin). On note souvent cette classe de problème par VSBPP (variable sized bin packing problem).

Il est important de savoir si l'on a le droit de tourner l'objet en question (cela ne concerne évidemment que les problèmes d'au moins deux dimensions), par exemple, on trouve trois cas possibles pour le cas à deux dimensions [60] :

1. Aucune orientation n'est permise, on parle du cas orienté,
2. On peut tourner l'objet d'un angle de 90°,
3. Toute orientation permise, on peut faire tourner l'objet suivant n'importe quel angle.

Disponibilité des objets et bins

La disponibilité des objets peut être sujette à un ordre donné. Si les objets arrivent un par un, sans connaissance à priori des suivants, on parle alors de la version en-ligne. Par contre, on parle de la version hors-ligne, si tout les objets sont connus avant leur rangement. Le nombre de bins utilisés peut être contraint par des bornes inférieures et supérieures, dans le problème de Strip packing deux dimensions, il faut placer les objets dans un seul rectangle de longueur infinie de manière à minimiser la longueur utilisée.

Contraintes sur le rangement

Différentes contraintes existent sur le rangement $\mathcal{P}$. On peut citer les plus importantes comme :

- l'ordre dans lequel les objets doivent être retirés du bin,
- contrainte de poids (le poids total d'un bin ne peut excéder une certaine limite),
- contrainte de conflit entre objets,
- contrainte de fragilité, les objets résistants doivent être placés sous les objets fragiles,
- contrainte guillotine (2D-BPP), les objets doivent être restitués par des coupes côte-à-côte.

2.2 Problèmes de bin packing unidimensionnels

Le problème de bin packing à une dimension peut être décrit par les données de n objets et m bins ($m \leq n$) avec

$$a_i = \text{taille de l'objet i,}$$

$$C = \text{capacité de chacun des bins,}$$

Le but étant d'assigner chaque objet dans un bin sans excéder la capacité du bin, sans fractionner les objets et en minimisant le nombre de bins utilisés.

2.2.1 Complexité du problème de bin packing

On donne ici quelques résultats importants [46] sur la complexité du bin packing.

Théorème 2.2.1. *Le problème suivant est NP-complet : étant donné une instance I du 1D-BPP, décider si I a une solution avec deux boites.*

Théorème 2.2.2. *Le problème suivant est fortement NP-complet : étant donné une instance I du 1D-BPP et un nombre m, décider si I peut être résolu avec B bins.*

Théorème 2.2.3. *(Théorème d'inapproximabilité). A moins que P=NP, il n'existe pas d'algorithme d'approximation de facteur k pour le 1D-BPP pour tout* $k < \frac{3}{2}$.

2.2.2 Formulation mathématique

Il existe plusieurs modélisations [95] à ce problème, nous en présentons les plus connues. Tout les modèles suivants supposent la connaissance complète des objets à ranger.

Modèle de Kantorovitch :

Considérons les variables de décisions suivantes :

$$y_j = \begin{cases} 1, & \text{si le bin j est utilisé ;} \\ 0, & \text{sinon.} \end{cases}$$

$$x_{ij} = \begin{cases} 1, & \text{si l'objet i est assigné au bin j ;} \\ 0, & \text{sinon.} \end{cases}$$

Le modèle mathématique est alors :

$$\min \sum_{j=1}^{n} y_j, \tag{2.1}$$

$$\sum_{i=1}^{n} a_i x_{ij} - C y_j \leq 0, \quad j = \overline{1,n}; \tag{2.2}$$

$$\sum_{j=1}^{m} x_{ij} = 1, i = \overline{1,n}; \tag{2.3}$$

$$y_j, \quad x_{ij} \in \{0,1\}, i = \overline{1,n}, \quad j = \overline{1,n}; \tag{2.4}$$

L'inéquation (2.2) indique que la taille totale des objets placés dans un bin n'excède pas sa capacité. L'équation (2.3) indique que chaque objet est placé dans exactement un bin. L'équation (2.4) indique qu'aucun objet ne peut être fractionné.

Remarquons la ressemblance avec le problème du sac-à-dos, la contrainte (2.2) est dite contrainte de type sac-à-dos. On pourrait penser que remplir au maximum sac par sac (résoudre une succession de problèmes de la somme de sous-ensembles, voir chapitre 1) permettra de résoudre le problème de bin packing. Considérons l'exemple suivant :

Exemple :

Soit à ranger les objets de tailles : 12, 11, 11, 7, 7, 6 dans un minimum de bins de capacité C=20. Si on remplit au maximum le premier bin, on rangera l'ensemble d'objets $\{7,7,6\}$*, puis les autres objets doivent être rangés dans des bins différents, on aura alors une solution à 4 bins. Or la solution optimale de ce problème est à 3 bins (*$\{12,7\},\{11,7\},\{11,6\}$*).*

L'exemple précédent nous montre une différence importante entre le problème de la somme de sous-ensembles et le bin packing.

Modèle à variables entières :

Une autre modélisation peut être donnée en considérant le nombre de fois où l'objet i apparaît, notons ce nombre n_i. En utilisant les variables suivantes :

- $y_j = \begin{cases} 1, & \text{si le bin j est utilisé ;} \\ 0, & \text{sinon.} \end{cases}$
- $x_{ij} = \begin{cases} k, & \text{si k occurrences de l'objet i sont assignés au bin j ;} \\ 0, & \text{sinon.} \end{cases}$

Le modèle s'écrit alors :

$$\min \sum_{j=1}^{n} y_j \tag{2.5}$$

$$\sum_{i=1}^{n} a_i x_{ij} - C y_j \leq 0, j = 1, ..., n \tag{2.6}$$

$$\sum_{j=1}^{n} x_{ij} = n_i, i = 1, ..., n \tag{2.7}$$

$$y_j \in \{0, 1\}, \quad x_{ij} \in \{0, 1, ..., n_i\} \forall i, j = 1, ..., n \tag{2.8}$$

Modèle de bin packing à une dimension avec conflits

Le problème de bin packing avec conflits est une variante du modèle classique dans lequel certains objets ne peuvent être mis ensemble dans un même bin. Le modèle suivant s'appuie sur le modèle de Kantorovitch. Rappelons les variables de décisions utilisées :

$$y_j = \begin{cases} 1, & \text{si le bin j est utilisé ;} \\ 0, & \text{sinon.} \end{cases}$$

$$x_{ij} = \begin{cases} 1, & \text{si l'objet i est assigné au bin j ;} \\ 0, & \text{sinon.} \end{cases}$$

Pour rendre compte des contraintes de conflits, considérons deux objets a_{i_1}, a_{i_2} en conflit, alors ces deux objets ne peuvent être mis dans le même bin. Il s'ensuit alors les contraintes suivantes :

$$x_{i_1 j} + x_{i_2 j} \leq 1, \forall j \in \{1, ..., n\}.$$

Soit $I = \{(l, k), l, k \in \{1, ..., n\}\}$, l'ensemble des couples d'objets en conflits, i.e. : a_l et a_k sont en conflit pour tout $(l, k) \in I$.

Le modèle du problème de bin packing à une dimension avec conflit s'énonce alors :

$$\min \sum_{j=1}^{n} y_j, \tag{2.9}$$

$$\sum_{i=1}^{n} a_i x_{ij} - C y_j \leq 0, \quad j = \overline{1, n}; \tag{2.10}$$

$$\sum_{j=1}^{m} x_{ij} = 1, i = \overline{1, n}; \tag{2.11}$$

$$x_{lj} + x_{kj} \leq 1, j = \overline{1, n}, (l, k) \in I; \tag{2.12}$$

$$y_j, \quad x_{ij} \in \{0, 1\}, i = \overline{1, n}, \quad j = \overline{1, n}; \tag{2.13}$$

2.2.3 Heuristiques

Dans cette section, on va présenter quelques heuristiques classiques ainsi que leur performances. On peut trouver dans [16; 28; 33; 44; 63; 64] d'autres heuristiques pour le 1D-BPP et ses variantes.

Notation :

1. I : une instance du problème de bin packing.
2. OPT(I) : le nombre minimal de bin suffisant pour le rangement de tous les objets.
3. $\mathfrak{A}$(I) : le nombre de bin utilisé par l'algorithme $\mathfrak{A}$ pour ranger tous les objets.

Heuristiques pour le 1D-BPP

L'algorithme le plus simple de type glouton est sûrement l'algorithme Next-fit (ajustement suivant), c'est aussi un algorithme en ligne. Il place le premier objet dans le bin 1 puis, chaque objet suivant est placé dans le bin courant si possible, sinon un nouveau bin est ouvert et devient ainsi le bin courant.

Algorithme 1 : Next-fit ($\mathfrak{NF}$)

Entrée : liste d'objets $I = \{a_1, a_2, ..., a_n\}$.

Sortie : une affectation f des objets dans les bins et le nombre de bins utilisés.

1. poser $k = 0$, $S = 0$;
2. **pour** $i = 1$ à n **faire**

   si $S + a_i > C$ **alors** poser $k = k + 1$, $S = 0$, **finsi**

   $f(i) = k$, $S = S + a_i$.

   finpour

Théorème 2.2.4. *[46] L'algorithme Next-fit s'exécute en O(n). Pour toute instance* $I = \{a_1, ..., a_n\}$, *on a :*

$$\mathfrak{NF}(I) \leq 2\lceil \frac{\sum_{i=1}^{n} a_i}{C} \rceil - 1 \leq 2OPT(I) - 1. \tag{2.14}$$

Proposition 2.2.1. *[46] Soit* $0 < \gamma < 1$, *et C==1. Pour toute instance* $I = \{a_1, ..., a_n\}$ *telle que* $a_i < \gamma \quad \forall i \in \{1, ..., n\}$, *on a*

$$\mathfrak{NF}(I) \leq \lceil \frac{\sum_{i=1}^{n} a_i}{1 - \gamma} \rceil. \tag{2.15}$$

L'algorithme suivant, dit de la première zone libre, est une amélioration du précédent algorithme qui gâche l'espace libre des bins déjà utilisés. Dans cet algorithme on place l'objet dans le premier bin possible, sinon on ouvre un nouveau.

Algorithme 2 : First-fit ($\mathfrak{FF}$)

Entrée : liste d'objets $I = \{a_1, a_2, ..., a_n\}$.

Sortie : une affectation f des objets dans les bins et le nombre de bin utilisé.

1. **pour** $i = 1$ à n **faire**

$$f(i) = \min\{j \in \mathbb{N} : \sum_{h<i : f(h)=j} a_h + a_i \leq C\}.$$

   finpour

2.

$$k = \max_{i \in \{1,\dots,n\}} f(i).$$

L'algorithme First-fit est évidemment aussi bon que Next-fit, il est donc aussi un algorithme d'approximation de facteur 2.

Théorème 2.2.5. *[46] Pour toute instance I, on a :*

$$\mathfrak{FF}(I) \leq \lceil \frac{17}{10} OPT(I) \rceil. \tag{2.16}$$

Une autre variante, dite best-fit, consiste à placer chaque objet dans le bin dont l'espace restant est minimal.

Algorithme 3 : Best-fit ($\mathfrak{BF}$)

Entrée : liste d'objets $a_1, a_2, ..., a_n$.

Sortie : une affectation f des objets dans les bins et le nombre de bin utilisé.

1. **pour** $i = 1$ à n **faire**

$$J = \{j \in \mathbb{N} : \sum_{h<i : f(h)=j} a_h + a_i \leq C\}.$$

$$f(i) = \arg\min_{j \in J} (c - \sum_{f(h)=j} a_h).$$

finpour

2.

$$k = \max_{i \in \{1,\dots,n\}} f(i).$$

La proposition 2.2.1 montre que l'algorithme Next-fit (à fortiori First-fit) se comporte d'autant mieux que les objets sont petits, il est donc intéressant de placer les plus gros en premier, les autres étant de plus en plus faciles à placer dans les bins déjà ouverts, ceci implique la connaissance de tous les objets à ranger, ce qui nous conduit à un algorithme hors-ligne. Il s'agit donc de trier les objets par ordre décroissant de longueur puis d'appliquer un des algorithmes précédents, on obtient ainsi le Next-fit decreasing, First-fit decreasing et le Best-fit decreasing.

Théorème 2.2.6. *[88] L'algorithme First-fit decreasing ($\mathfrak{FFD}$) fournit une approximation de facteur $\frac{3}{2}$.*

Théorème 2.2.7. *[101] Pour toute instance I, on a :*

$$\mathfrak{FFD}(I) \leq \frac{11}{9}OPT(I) + 1. \tag{2.17}$$

Heuristiques pour le 1D-BPP avec conflits

Les heuristiques dédiées au problème de bin packing avec conflits sont basées sur celles du 1D-BPP. Considérons les heuristiques présentées plus haut, pour les adapter au cas avec conflit, il suffit de vérifier la contrainte de conflit, en plus de la contrainte de capacité. Considérons à titre d'exemple l'heuristique du $\mathfrak{FFD}$, la version avec conflit $\mathfrak{FFDC}$ peut être schématisé comme suit : **Algorithme $\mathfrak{FFDC}$**

- Entrée : liste d'objet $\{a_1, ..., a_n\}$ et un ensemble de conflits.
- Sortie : une solution réalisable.

1. Trier les objets par ordre décroissant de longueur.
2. Pour i=1 à n faire
   - Pour j=1 à n,
     - Ranger l'objet k dans le bin j, s'il ne provoque pas de conflit et s'il y'a suffisamment d'espace.
   - Finpour.
- Finpour.

2.2.4 Bornes inférieures

Dans cette section, on présente les différentes techniques pour l'obtention de bornes inférieures pour le 1D-BPP classique.

Définition 2.2.1. *Soit I une instance d'un problème de minimisation, OPT(I) la valeur de l'objectif à l'optimum. L(I) est une borne inférieure si elle vérifie l'inégalité*

$$L(I) \leq OPT(I)$$

En général, pour un problème de minimisation, on cherche une procédure permettant de calculer une borne inférieure pour une instance quelconque. Une borne inférieure doit

évidemment être la plus proche possible de l'optimum, et une procédure est meilleure qu'une autre si elle permet de fournir une borne inférieure plus grande. On s'interesse aussi à la performance individuelle d'une procédure, en mesurant l'écart entre la borne inférieure et l'optimum, au pire cas.

Définition 2.2.2. *Soit $\mathfrak{P}$ un problème de minimisation et $E = \{I_1, I_2, ...\}$ l'ensemble des instances de ce problème, L_1, L_2 deux procédures fournissant des bornes inférieures au problème $\mathfrak{P}$. On dit que L_1 domine L_2 si :*

$$L_1(I) \geq L_2(I), \forall I \in E.$$

Définition 2.2.3. *Soit L une procédure fournissant une borne inférieure pour un problème de minimisation P, soit pour l'instance I, L(I) et Z(I) la valeur produite par L et la valeur optimale respectivement. Le ratio de performance au pire cas de L est défini comme le plus grand nombre $\rho(L)$ tel que :*

$$\rho(L) = \sup\{\frac{L(I)}{Z(I)},\ I \text{ est une instance de } P.\}$$

Le ration de performance asymptotique au pire cas de L est défini comme :

$$\rho_\infty(L) = \lim_{s \to +\infty} \sup\{\frac{L(I)}{Z(I)},\ I \text{ est une instance de } P \text{ tel que } OPT(I) \geq s.\}$$

Borne continue [66]

La borne continue L_0 est calculée en sommant les longueurs des objets divisé par la longueur du bin.

$$L_0 = \lceil \frac{\sum_{i=1}^{n} a_i}{C} \rceil. \tag{2.18}$$

Evidemment plus les articles sont petits, plus l'évaluation est meilleure. Martello et Vigo [66] ont montré qu'elle pouvait atteindre $\frac{1}{2}$OPT dans le pire cas. Le défaut de cette borne est qu'elle ne considère pas les incompatibilités entre les grands objets qui ne peuvent être mis ensemble.

Borne de Martello et Vigo [15]

Martello et Vigo proposent une borne L^{MV}. L'idée est de mettre à part les articles qui ne peuvent être dans le même bin. Le calcul de cette borne est basé sur une décomposition

de l'ensemble des articles en trois sous-ensembles dépendant d'un paramètre k : les grands articles (A_{gr}), les articles moyens (A_{moy}), et les petits articles (A_{pt}). Soit k un entier, $1 \leq k \leq \frac{1}{2}C$:

- $A_{gr} = \{a_i \in I : a_i > c - k\}$,
- $A_{moy} = \{a_i \in I : C - k \geq a_i > \frac{1}{2}C\}$,
- $A_{pt} = \{a_i \in I : \frac{1}{2}C \geq a_i \geq k\}$.

Les articles de taille inférieure à k sont ignorés. Il est évident que les grands articles ainsi que les moyens ne peuvent être rangés ensemble dans le même bin, il s'ensuit alors cette première borne L_α, dont le terme à droite est une évaluation par la borne continue, du nombre de bins qu'il faut pour ranger les petits objets avec les moyens :

$$L_\alpha = \max_{1 \leq k \leq \frac{1}{2}C} \{|A_{gr} \cup A_{moy}| + \max\{0, \lceil \frac{\sum_{a_i \in A_{moy} \cup A_{pt}} a_i}{C} - |A_{moy}| \rceil\}\} \tag{2.19}$$

Une deuxième borne L_β tient compte du nombre de petits objets qui ne peuvent être rangés avec les moyens.

$$L_\beta = \max_{1 \leq k \leq \frac{1}{2}C} \{|A_{gr} \cup A_{moy}| + \max\{0, \lceil \frac{|A_{pt}| - \sum_{a_i \in A_{moy}} \lfloor \frac{C - a_i}{k} \rfloor}{\lfloor \frac{C}{k} \rfloor} \rceil\}\} \tag{2.20}$$

La borne de Martello et Vigo est donnée par :

$$L^{MV} = \max\{L_\alpha, L_\beta\}. \tag{2.21}$$

Bornes de Fekete et Scheppers [27]

Fekete et Scheppers utilisent les fonctions dual-réalisable (DFF) pour calculer des évaluations par défaut pour le 1D-BPP.

Définition 2.2.4. *Soit $g : [0,1] \to [0,1]$, g est dite dual-réalisable (DFF), si pour tout ensemble fini S de réels, on a :*

$$\sum_{x \in S} x \leq 1 \Rightarrow \sum_{x \in S} g(x) \leq 1 \tag{2.22}$$

Les DFF s'appliquent sur des instances où la longueur du bin est unitaire et celles des objets comprises entre 0 et 1. Le théorème suivant montre comment utiliser les DFF :

Théorème 2.2.8. *Soit $I = \{a_1, a_2, ..., a_n\}$ une instance du 1D-BPP. Soit f une DFF. Alors, toute borne inférieure pour le problème transformé $f(I) = \{f(a_1), f(a_2), ..., f(a_n)\}$ est aussi une borne inférieure pour l'instance I.*

Plusieurs DFF ont été proposées par Fekete et Scheppers [27] :

Théorème 2.2.9. *Soit $k \in \mathbb{N}$, $\epsilon \in [0, \frac{1}{2}]$. Les fonctions $u^{(k)}, U^{(\epsilon)}, \phi^{(\epsilon)} : [0,1] \to [0,1]$ définies ci-après sont des fonctions dual-réalisables :*

- $$u^{(k)}(x) = \begin{cases} x, & si\ x(k+1) \in \mathbb{Z}\,; \\ \lfloor (k+1)x \rfloor \frac{1}{k}, & sinon. \end{cases}$$
- $$U^{(\epsilon)}(x) = \begin{cases} 1, & si\ x > 1-\epsilon\,; \\ x, & si\ \epsilon \leq x \leq 1-\epsilon\,; \\ 0, & si\ x < \epsilon. \end{cases}$$
- $$\phi^{(\epsilon)}(x) = \begin{cases} 1 - \frac{\lfloor (1-x)\epsilon^{-1} \rfloor}{\lfloor \epsilon^{-1} \rfloor}, & si\ x > \frac{1}{2}\,; \\ \frac{1}{2}, & si\ x = \frac{1}{2}\,; \\ \frac{1}{\lfloor \epsilon^{-1} \rfloor}, & \epsilon \leq x < \frac{1}{2}\,; \\ 0, & x < \epsilon. \end{cases}$$

Remarquons que la seconde DFF permet de retrouver la borne de Martello et Toth L_2 [27], dont la performance au pire cas est de $\frac{2}{3}$:

$$L_2(I) = \max_{\epsilon \in [0, \frac{1}{2}]} (U^{(\epsilon)}(I)) = \max_{\epsilon \in [0, \frac{1}{2}]} (|\{a_i \in I : a_i > 1-\epsilon\}| + L_0(\{a_i \in I : \epsilon \leq a_i \leq 1-\epsilon\})) \tag{2.23}$$

Fekete et Scheppers [27] construisent deux bornes en faisant une mixture de DFF. Pour $k \in \mathbb{N}$,

$$L_2^{(k)} = \max_{\epsilon \in [0, \frac{1}{2}]} L_1(u^k \circ U^{(\epsilon)}(I)) \tag{2.24}$$

Pour $p \geq 2$,

$$L_*^{(p)} = \max\{L_2(I), \max_{k=2,...,p} L_2^{(k)}(I)\} \tag{2.25}$$

Ils montrent que la borne L_*^2 est égale à l'optimum quand la taille des objets est supérieures ou égale au tiers de celles des bins. En outre ils prouvent que les performances asymptotiques au pire cas des bornes $L_*^{(2)}$ et $L_*^{(p)}$ sont égales à $\frac{3}{4}$.

2.2.5 Technique de réduction

On entend par réduction l'ensemble des affectations d'objets effectuées sans risquer l'optimalité de la solution, cette technique de réduction est basée sur le critère de dominance de Martello et Toth [66].

Définition 2.2.5. *On dit qu'un ensemble $F \subseteq I$ est réalisable si $\sum_{i \in F} a_i \leq C$.*
Etant donné deux ensembles réalisables F_1 et F_2, on dit que F_1 domine F_2 si la valeur optimal de la solution obtenue en affectant les objets de F_1 au bin i^*, n'est pas supérieur à la valeur obtenue en affectant ceux de l'ensemble F_2.

Théorème 2.2.10. ***Critère de dominance****[66] Soient F_1 et $F2$ deux ensemble réalisable, s'il existe une partition de F_2 en sous-ensembles P_1,...,P_l et un sous-ensemble $\{j_1, ..., j_l\}$ de F_1 tel que :*

$$a_{j_h} \geq \sum_{k \in P_h} a_k, \quad \forall h = 1, ..., l$$

alors F_1 domine F_2.

Il est alors possible d'affecter certain objet d'office s'il constitue un ensemble dominant tout autre ensemble d'objet et pourant être supprimé du problème. Comparer toute les combinaisons possible est toute fois impossible à faire. Martello et Toth propose une procédure appellée MTRP testant les ensembles dont la cardinalité ne dépasse pas 3. On considère que les objets sont triés de manière décroissante. On obtient le nombre de bins remplis de manière optimale, et pour chacun des objets $j \in N$:

- $$b_j = \begin{cases} 0, & \text{si l'objet j n'est pas assigné ;} \\ k, & \text{où k est l'indice du bins dans lequel l'objet j est rangé.} \end{cases}$$
- z^r le nombre de bins dans lesquels on a rangé des objets, qui sont fermés.

Procédure 1 : MTRP [66] :

Données n, a_j, C ;

Résultat z^r, (b_j).

Début de la procédure

$N = \{1, ..., n\}$;

$\overline{N} = \{\}$;

$z^r = 0$;

Pour j=1 **à** n **faire** $b_j = 0$.

Repeter Trouver $j = \min\{h : h \in N \backslash \overline{N}\}$;

Soit $N' = N \backslash \{j\} = \{j_1, ..., j_l\}$ avec $w_{j_1} \geq ... \geq w_{j_l}$;

$F = \{\}$;

Trouver le plus grand k tel que $w_j + \sum_{q=l-k+1}^{l} w_{j_q} \leq C$;

Si k=0 **alors** $F = \{j\}$;

sinon $j^* = \min\{h \in N' : w_j + w_h \leq C\}$;

Si k=2 **alors**

Trouver $j_a, j_b \in N'$ avec $a < b$, tel que : $w_{j_a} + w_{j_b} = \max\{w_{j_r} + w_{j_s} : j_r, j_s \in N' : w_j + w_{j_r} + w_{j_s} \leq C\}$;

Si $w_{j^*} \geq w_{j_a} + w_{j_b}$ **alors** $F = \{j, j^*\}$

Sinon si $w_{j^*} = w_{j_a}$ et $(b - a \leq 2$ ou $w_j + w_{j_b-1} + w_{j_b-2} > C)$ alors $F = \{j, j_a, j_b\}$.

Si $F = \{\}$ **alors** $\overline{N} = \overline{N} \bigcup \{j\}$;

Sinon $z^r = z^r + 1$;

pour tout $h \in F$ **faire** $b_h = z^r$;

$N = N \backslash F$; ;

Jusqu'à $(N \backslash \overline{N} = \{\})$

Fin de la procédure

2.2.6 Algorithmes exacts

Les algorithmes exacts traitant du bin packing sont assez peu nombreux par rapport aux heuristiques et métaheuristiques, nous en énumérons quatre algorithmes de types

séparation et évaluation, deux se basent sur la programmation linéaire et deux résolvent le problème par une succession d'affectations d'objets.

Algorithme MTP (Martello and Toth procedure)

Le premier algorithme dédié est celui de Martello et Toth (MTP) [66] de type SE, qui affecte les objets tout en évitant les cas dominés, l'exploration est faite en profondeur d'abord et à chaque étape on calcule la borne inférieure L_2 en considérant les objets affectés dans un bin comme étant un seul objet, puis une solution est calculée par l'algorithme du best-fit et du first fit.

Algorithme BISON

Scholl et al. [85] ont introduit un algorithme exact pour la résolution du 1D-BPP, on peut considerer BISON comme une amélioration de MTP. L'amélioration se situe au niveau du calcul des bornes inférieures, utilisation de plusieurs heuristiques en appui et utilisation de plusieurs critères de dominance, de plus l'algorithme peut affecter plusieurs objets dans un même bin si cela forme un ensemble non dominé par un autre, contrairement à MTP qui affecte objet par objet.

Algorithme de Korf

L'algorithme de Korf nommé "Bin completion algorithm" [45] procède de manière similaire que pour l'algorithme MTP à ceci près qu'il utilise trois critère de dominance afin de remplir le mieux possible les bins ouverts. Il applique son algorithme sur des instances de 100, 500 et 1000 objets de Fekete et Scheppers. Les résultats sont comparés à ceux obtenus par l'algorithme MTP, mais reste prohibitif, à partir de 40 objets, l'algorithme nécessite en moyenne plus de 60 secondes.

Algorithme de Valério

L'algorithme de Valério [96] est certainement le plus efficace jusqu'à présent, il se base sur une formulation assez complexe du problème de bin packing basé sur la théorie des

graphes, cette formulation lui permet d'éviter des cas symétriques et améliore considérablement la convergence de l'algorithme, de plus il s'agit d'un algorithme de branch-and-price. Sur toutes les instances entières de Falknauer, l'algorithme donne la solution optimale après 5 secondes en moyenne, sans grand écart sur les différentes instances, à ce titre l'algorithme de Valério est certainement le plus robuste des algorithmes de résolution du bin packing.

2.3 Problèmes de bin packing bidimensionnels

Le problème de bin packing à deux dimensions est défini par la donnée de n objets rectangulaires de longueur h_j, et de largeur w_j, $j = 1, ..., n$, à ranger dans de grands rectangles (bins) de longueur H et de largeur W, en utilisant le moins possible de bins. Généralement, sauf indication contraire les cotés des objets doivent être parallèles à ceux du bin. Ce problème a de multiples applications pratiques comme la coupe de plaque de taille standard en rectangles de diverses dimensions, en industrie du bois ou du verre, placement d'articles dans un journal.
Une variante trés étudiée du 2D-BPP dite strip-packing, consiste à placer tous les objets dans un seul bin de longueur infinie, de manière à minimiser la longueur utilisée.
Mise à part le cas spécial du strip-packing, on distingue deux types de contraintes majeures rencontrées dans la litérature, donnant quatre types de problème classique du 2D-BPP qui sont :

1. Orientation : les objets peuvent être tournés ou non suivant un angle de 90°.
2. Guillotine : Si elle est imposée, les objets sont obtenus suite à une coupe côte-à-côte parrallèlement aux cotés du bin.

La contrainte guillotine est fréquemment présente dans les problèmes de coupe (cutting), cela est dû aux caractéristiques techniques des machines faisant la coupe. La rotation par contre est par exemple exclue dans les problèmes de placement d'articles journalistiques. Remarquons que le cas où la largeur des bins est égale à celles des objets on retrouve alors le 1D-BPP, comme le 1D-BPP est fortement NP-Complet, le 2D-BPP l'est forcément.

2.3.1 Formulation mathématique

Etant donné que le problème à deux dimensions a au moins quatre variantes, il n'existe pas de formulation unique. Gilmore et Gomory ont proposé le premier modèle pour le 2D-BPP(sans guillotine, orientation fixe), de la même manière que pour le 1D-BPP dans une approche par génération de colonnes.

Modèle pour le problème de bin packing deux dimensions :

Dans ce problème [71], l'objectif est de ranger tous les objets dans un nombre minimal de bins et de manière parallèle à leur cotés. Les auteurs supposent en plus que le rangement doit respecter les contraintes suivantes :

1. Les objets sont rangés par niveau, un niveau est délimité par l'arête supérieure de l'objet ayant la plus grande longueur rangée dans ce niveau.
2. A chaque niveau, l'objet le plus à gauche est le plus grand.
3. Sur chaque bin, le premier niveau est le plus grand.

Les objets sont définis par une largeur $w_i, i = \overline{1,n}$ et une longueur $h_i, i = \overline{1,n}$, ils sont considérés dans l'ordre décroissant de leur longueur. Il peut être modélisé en supposant qu'il y'a n niveaux potentiels (le i^{eme} niveau est associé au i^{eme} objet qui l'initialise), et n bins potentiels (le k^{eme} bin est associé au k^{eme} niveau qui l'initialise).
Soit $J = \{1, ..., n\}$ et soit les variables de décisions suivantes :

- $y_i = \begin{cases} 1, & \text{si l'objet i initialise le niveau i, } i \in J\,; \\ 0, & \text{sinon.} \end{cases}$
- $q_k = \begin{cases} 1, & \text{si le niveau k initialise le bin k, } k \in J\,; \\ 0, & \text{sinon.} \end{cases}$
- $x_{ij} = \begin{cases} 1, & \text{si l'objet j est rangé au niveau i, } i \in J \backslash N \text{ et j>i}\,; \\ 0, & \text{sinon.} \end{cases}$
- $z_{ki} = \begin{cases} 1, & \text{si le niveau i est alloué au bin k}\,; \\ 0, & \text{sinon.} \end{cases}$

Le modèle s'écrit alors :

$$\min \sum_{k=1}^{n} q_k \tag{2.26}$$

$$\sum_{i=1}^{j-1} x_{ij} + y_j = 1, \quad j = 1, ..., n. \tag{2.27}$$

$$\sum_{j=i+1}^{n} w_j x_{ij} \leq (W - w_i) y_i, \quad i = 1, ..., n-1. \tag{2.28}$$

$$\sum_{k=1}^{i-1} z_{ki} + q_i = y_i, \quad i = 1, .., n. \tag{2.29}$$

$$\sum_{i=k+1}^{n} h_i z_{ki} \leq (H - h_k) q_k, \quad k = 1, ..., n-1. \tag{2.30}$$

$$q_k, y_i, x_{ij}, z_{zi} \in \{0, 1\} \forall i, j, k \in J \tag{2.31}$$

Les restrictions j>i et i>k sont dus aux hypothèses 1-3 du problème. Les formules 2.27 (resp. 2.29) impose que chaque objet est rangé une seule fois (resp. chacun des niveau est affecté à un seul bin). Les formules 2.28 et 2.30 imposent de respecter la largeur de chacun des niveaux ainsi que la hauteur des bins.

Modèle pour le problème de Strip packing :

Dans ce problème [71], on a un seul bin de largeur fixé à W et d'une longueur infinie, l'objectif étant de ranger tous les objets sur une longueur minimale. Le modèle peut se déduire directement du modèle précédent en ne gardant que les variables x_{ij}, y_i ainsi que les deux premières contraintes :

$$\min \sum_{i=1}^{n} h_i y_i \tag{2.32}$$

$$\sum_{i=1}^{j-1} x_{ij} + y_j = 1, \quad j = 1, ..., n. \tag{2.33}$$

$$\sum_{j=i+1}^{n} w_j x_{ij} \leq (W - w_i) y_i, \quad i = 1, ..., n-1. \tag{2.34}$$

$$x_{ij}, y_i \in \{0, 1\} \forall i, j \in J \tag{2.35}$$

Modèle pour le problème de découpe en deux dimensions sans guillotine :

Dans ce problème de découpe [69], on a un seul bin de longueur L et de largeur W, chaque objet est défini par une longueur h_i et une largeur w_i, il figure b_i fois et a une valuation v_i, l'objectif étant de maximiser la valuation totale des objets découpés dans le bin. Le modèle se base sur les élément suivant :

- $x_{ipq}=\begin{cases} 1, & \text{si l'objet de type i est placé à la position (p,q)}\,; \\ 0, & \text{sinon.} \end{cases}$
  P = {p : p=$\sum_{i=1} m\alpha_i h_i, p \leq L - \min\{h_i, i = 1, ..., m\}, \alpha_i \in \mathbb{N}\}$.
- $Q = \{q : q = \sum_{i=1} m\beta_i w_i, q \leq W - \min\{w_i, i = 1, ..., m\}, \beta_i \in \mathbb{N}\}$.
- $P_i = \{p : p \in P, p \leq L - l_i\}$.
- $Q_i = \{q : q \in Q, q \leq W - w_i\}$.
- $a_{ipqrs} = \begin{cases} 1, & \text{si } p \leq r \leq p + l_i - 1 \text{ et } q \leq s \leq q + w_i - 1\,; \\ 0, & \text{sinon.} \end{cases}$

Le modèle s'écrit :

$$\max \sum_{i=1}^{m} \sum_{p \in P_i} \sum_{q \in Q_i} v_i x_{ipq} \tag{2.36}$$

$$\sum_{i=1}^{m} \sum_{p \in P_i} a_{ipqrs} x_{ipq} \leq 1 : \forall r \in P, s \in Q, \tag{2.37}$$

$$\sum_{p \in P_i} \sum_{q \in Q_i} x_{ipq} \leq b_i, \forall i \in \{1, ..., m\} \tag{2.38}$$

$$x_{ipq} \in \{0, 1\}, \forall i, p, q. \tag{2.39}$$

La formule 2.37 indique que si un rangement d'un objet est effectué alors il ne chevauche pas un autre objet. La formule 2.38 indique que le nombre d'objets rangés d'un type donné ne peut dépasser le nombre de son occurrence.

2.3.2 Heuristiques

Les algorithmes qu'on présente ici sont de type hors-ligne. Ils sont utilisés pour le 2D-BPP (orientation fixe, sans guillotine) et ils se scindent en deux familles :

- **Algorithmes en une phase** : on range les objets directement dans les bins ;
- **Algorithmes en deux phases** : on commence par placer les objets dans un seul bin imaginaire de longueur infinie, puis en second lieu on obtient une solution en découpant le bin en plusieurs autres de longueurs n'excédant pas la longueur des bins.

De plus, la majorité de ces algorithmes procèdent par niveau, i.e. un rangement est obtenu par placement des objets, de gauche à droite, sur des lignes formant des niveaux. Le premier niveau est l'arête inférieur du premier bin utilisé, puis le nouveau niveau est produit par l'arête supérieure de l'objet le plus grand. Il existe trois stratégies classiques pour résoudre le problème du Strip packing, issues des algorithmes classiques du cas à une dimension. Les objets sont préalablement triés par ordre décroissant de longueur. Soit j l'objet courant, et s le dernier niveau créé :

1. **Next-Fit Decreasing Height** ($\mathcal{NFDH}$) : l'objet j est placé le plus à gauche du niveau s, si possible, sinon un nouveau niveau ($s = s + 1$) est créé et l'objet j est placé à gauche ;
2. **First-Fit Decreasing Height** ($\mathcal{FFDH}$) : l'objet j est placé le plus à gauche du niveau de plus petit indice, s'il n'y a pas de niveau pouvant le contenir, on initialise un nouveau niveau ;
3. **Best-Fit Decreasing Height** ($\mathcal{BFDH}$) : On choisi le niveau dans lequel on place l'objet j, de manière à minimiser l'espace horizontal restant, s'il n'y a pas de niveau pouvant le contenir, on en initialise un ;

Coffman et al. [17] ont analysé les performance du $\mathcal{NFDH}$ et $\mathcal{FFDH}$ pour le Strip packing deux dimensions. Ils prouvent en outre, sous condition $\max_j\{h_j\} = 1$, que :

$$\mathcal{NFDH}(I) \leq 2.OPT(I) + 1 \tag{2.40}$$

$$\mathcal{FFDH}(I) \leq \frac{17}{10}.OPT(I) + 1 \tag{2.41}$$

De plus les bornes supérieurs ainsi obtenues sont les plus petites possibles ; et si les hauteur ne sont pas normalisés, alors les coefficients ne sont pas affectés. Les trois algorithmes s'exécute en $O(n \log n)$.

Algorithmes en deux phases

Un algorithme appelé **Hybrid First-Fit** ($\mathcal{HFF}$) a été proposé par Chung et al. [16] Dans la première phase une solution au strip packing est obtenue par l'algorithme $\mathcal{FFDH}$. Soit $H_1, H_2, ...$ les hauteurs des niveaux ainsi obtenus. On remarque que $H_1 \geq H_2 \geq ...$, puis une solution au bin packing est obtenue en résolvant par une heuristique le problème de bin packing à une dimension (les objets sont de tailles H_i et H la capacité des bins) par l'algorithme First-Fit-Decreasing. La complexité de l'algorithme est de $O(n \log n)$. Quand aux performances de l'algorithme, Chung et al. prouve que :

$$\mathcal{HFF}(I) \leq \frac{17}{8}.OPT(I) + 5. \tag{2.42}$$

Une autre variante due à Berkey et Wang [4] appellé **Finite Best Strip** ($\mathcal{FBS}$), est basée sur l'algorithme BFDH pour la première phase, puis sur le Best-Fit Decreasing pour la seconde phase. L'algorithme a été évalué expérimentalement sans que ses auteurs ne donnent une garantie de performance.
Considérons maintenant une autre variante du HFF appellé **Hybrid Next Fit** ($\mathcal{HNF}$), cette fois-ci basé sur le $\mathcal{NFDH}$ pour la première phase, et sur le Next-Fit Decreasing pour la seconde phase. Frenk et Galambos [30] ont analysé cet algorithme de complexité $O(n \log n)$ et donne la majoration suivante :

$$\mathcal{HNF}(I) \leq 3.382...OPT(I) + 9 \tag{2.43}$$

Loddi et al. [59] ont proposé l'algorithme **Floor-Ceiling** ($\mathcal{FC}$). En considérant une couche donnée, la ligne supérieure de l'objet le plus haut de cette couche est appelé "Ceiling" (plafond), et la couche inférieure "Floor" (plancher). Dans la première phase un objet est placé soit au plafond (de droite à gauche), soit au plancher (de gauche à droite), la première pièce placé au plafond doit être celle qui ne peut être posée sur le plancher. Le choix de la pièce à placer se fait suivant une stratégie de Best-Fit. si aucune pièce ne peut être placé dans la couche courante, une nouvelle couche est initialisée.

Dans la seconde phase, on résout un problème de bin packing unidimensionnelle formée par les couches obtenues. Les couches ainsi obtenues ont la même largeurs que les bins, et de hauteurs inférieurs. Dans la seconde phase est utilisé l'algorithme Best-fit Decreasing, ou un algorithme exacte.

Algorithmes en une phase

Berkey et Wang [4] ont présenté deux algorithmes, **Finite Next Fit** ($\mathcal{FNF}$) et **Finite First Fit** ($\mathcal{FFF}$). Ces deux algorithmes procèdent de manière très similaire à ceux en deux phases, les niveaux sont toujours présents mais à l'intérieur des bins. Ainsi la différence réside dans le fait que la longueur du bin n'est plus infinie, mais égale à celle du bin, on range les objets par niveau quand cela est possible, si ce n'est pas le cas un nouveau bin est initialisé. Ces algorithmes sont basés sur ceux du Strip packing, le premier sur le $\mathcal{NFDH}$ et l'autre sur le $\mathcal{FFDH}$, de complexité $O(n \log n)$.
Deux autres algorithmes de Berkey et Wang [4] sont proposés, ne gérant pas de couches : l'algorithme **Finite Bbottom Left** ($\mathcal{FBL}$) et **Next Bottom Left** ($\mathcal{NBL}$). Dans le premier, les pièces sont triées par largeurs décroissantes, ensuite chaque pièce est placée le plus bas et le plus à gauche possible du bin d'indice minimum. Si ce n'est pas possible, un nouveau bin est ouvert. Dans le deuxième, on ne considère que le bin courant, on ne revient plus à un bin refermé.
Lodi et al. [59] ont proposé l'algorithme **Alternate Direction** ($\mathcal{AD}$), l'algorithme est initialisé avec un nombre L (L étant une borne inférieure pour le problème) de bins, puis commence par placer au bas de ces bins des articles sélectionnes suivant la règle de Best-Fit Decreasing (les objets sont triés par ordre décroissant de longueur) de gauche à droite. Dans la seconde phase, les objets restants sont rangés un par un, alternativement de gauche à droite puis de droite à gauche.

2.3.3 Bornes inférieures

Borne continue [15]

La borne continue L_0 est calculée suivant la surface totale des objets :

$$\lceil \frac{\sum_{j=1}^{n} w_j h_j}{WH} \rceil \tag{2.44}$$

Martello et Vigo déterminent la performance au pire cas qui est :

$$L_0(I) \geq \frac{1}{4}.OPT(I) \tag{2.45}$$

Bornes de Martello et Vigo [15]

Martello et Vigo généralisent leurs bornes du 1D-BPP pour le 2D-BPP. Ainsi on distingue les décomposition suivantes :
Ils tiennent le même raisonnement suivant la largeur puis la longueur des objets à ranger. Etant donné un entier q : $1 \leq q \leq \frac{1}{2}W$, soient les ensembles suivants :

$$K_1^w = \{j \in I : w_j > W - q\}, \tag{2.46}$$

$$K_2^w = \{j \in I : W - q \geq w_j > \frac{1}{2}W\}, \tag{2.47}$$

$$K_3^w = \{j \in I : \frac{1}{2}W \geq w_j \geq q\}. \tag{2.48}$$

Remarquons que les objets de $K_1^w \cup K_2^w$ ne peuvent être mis côte-à-côte. Une première borne est donc :

$$L_1^w = |K_1 \cup K_2|,$$

Remarquons aussi que les objets de K_3^w et K_1^w ne peuvent être mis ensemble dans le même bin, on complète la borne précédente avec la borne L_0 des objets de K_3^w et K_2^w :

$$L_2^W(q) = L_1^w + \max\{0, \lceil \frac{\sum_{j \in K_2 \cup K_3} w_j h_j - (HL_1^w - \sum_{j \in K_1 h_j})W}{WH} \rceil\} \tag{2.49}$$

De manière symétrique, on obtient la borne $L_2^H(q)$, en faisant le même raisonnement sur la longueur des objets.

La borne de Martello et Vigo est finalement obtenue en prenant le maximum entre les deux évaluations :

$$\max\{\max_{1\leq q\leq \frac{1}{2}W}\{L_2^w(q)\}, \max_{1\leq q\leq \frac{1}{2}H}\{L_2^h(q)\}\} \tag{2.50}$$

La borne ainsi obtenue domine la borne continue, et peut être calculée en $O(n^2)$.
Une autre borne est proposée, calculable en $O(n^3)$. On définit pour cela les ensembles suivants : étant donnée deux entiers k,l : $1 \leq k \leq \frac{1}{2}W$, $1 \leq l \leq \frac{1}{2}H$:

$$A_{gr} = \{i \in I : h_i > H - let w_i > W_k\} \tag{2.51}$$

$$A_{moy} = \{i \in I\ A_{gr} : h_i > \frac{1}{2}H et w_i > \frac{1}{2}W\} \tag{2.52}$$

$$A_{pt}^s = \{i \in I : \frac{1}{2}H \geq h_i \geq let\frac{1}{2}W \geq w_i \geq k\} \tag{2.53}$$

Pour calculer la nouvelle borne, il faut determiner le nombre de pièces de taille (k,l) qui peuvent être rangées dans le bin contenant un article a_i, noté $m(a_i, k, l)$:

$$m(a_i,k,l) = \lfloor\frac{H}{l}\rfloor\lfloor\frac{W-w_i}{k}\rfloor + \lfloor\frac{W}{k}\rfloor\lfloor\frac{H-h_i}{l}\rfloor - \lfloor\frac{H-h_i}{l}\rfloor\lfloor\frac{W-w_i}{k}\rfloor \tag{2.54}$$

$$L_3(k,l) = |A_{gr} \cup A_{moy}| + \max\{0, \lceil\frac{|A_{pt}^s| - \sum_{i\in A_{moy}} m(a_i,k,l)}{\lfloor\frac{H}{l}\rfloor\lfloor\frac{W}{k}\rfloor}\rceil\} \tag{2.55}$$

Aucune relation de dominance n'existe entre L_2 et L_3. L'évaluation finale proposée par Martello et Vigo est le maximum entre L_2 et L_3.

Bornes de Fekete et Scheppers

Fekete et Scheppers [15] utilisent les mêmes DFF que celles appliquées pour le 1D-BPP. Ils appliquent ces DFF sur chacunes des dimensions du problème pour obtenir un problème à une seule dimension. La borne de Fekete et Scheppers domine celle de Martello et Vigo.

Soit $r, s \in]0, \frac{1}{2}]$, et pour tout $i \in I$:

- $a_i^{(1)(r)} = u^{(1)}(w_i).U^{(r)}(h_i)$,

- $a_i^{(2)(r)} = U^{(r)}(w_i).u^{(1)}(h_i)$,
- $a_i^{(3)(r)} = u^{(1)}(w_i).\phi^{(r)}(h_i)$,
- $a_i^{(4)(r)} = \phi^{(r)}(w_i).u^{(1)}(h_i)$,
- $a_i^{(5)(r)} = w_i.U^{(r)}(h_i)$,
- $a_i^{(6)(r)} = U^{(r)}(w_i).h_i$,
- $a_i^{(7)(r,s)} = \phi^{(r)}(w_i).\phi^{(s)}(h_i)$.

La borne de Fekete et Scheppers est donnée par l'expression :

$$L_{FS} = \max\{\max_{0<r\leq\frac{1}{2},\ k=\overline{1,6}}\{\lceil\sum_{i\in I} a_i^{(k)(r)}\rceil\}, \max_{0<s\leq\frac{1}{2},\ 0<r\leq\frac{1}{2}}\{\lceil\sum_{i\in I} a_i^{(7)(r,s)}\rceil\}\} \tag{2.56}$$

2.3.4 Algorithmes exacts

Les modèles définis dans les section 2.3.1 et 2.3.1 sont résolus par un algorithme exact [71]. Les problèmes sont reformulés sous forme de programmes DC puis résolus par un algorithme de coupe basé sur la reformulation DC. L'algorithme a été testé sur des instances générées aléatoirement, le nombre d'objets varie de 200 à 1000, et le nombre de variables de 20100 à 640800. Les problèmes sont résolus à l'optimum en moins de 23 secondes, en 2 itérations et 1 coupe.

Quand au problème présenté dans la section 2.3.1, l'auteur formule le problème sous forme d'un programme DC, puis résolu par l'algorithme DCA (voir chapitre 3). Les solutions obtenus ne sont pas toujours optimales car l'étude porte seulement sur l'efficacité du point initial de l'algorithme DCA [69].

Clautiaux [15] propose un algorithme exact basé sur une recherche arborescente, le problème traité est la version à deux dimensions avec orientation fixe, l'algorithme est testé sur des instances de 20 à 100 objets, avec une durée d'exécution maximale de 15 minutes. Les solutions obtenues sont de moins en moins bonnes à mesure que le nombre d'objets augmente. Ainsi pour 100 objets, seules 72% des instances traitées sont résolus à l'optimum.

Chapitre 3

Programmation DC et algorithme DCA

3.1 Introduction

Dans le domaine de l'optimisation globale, la programmation DC joue un rôle important de part son aspect théorique et le large spectre d'applications traitées. Une fonction est dite DC si elle peut être représentée comme différence de deux fonctions convexes. Un problème de programmation mathématique avec des fonctions DC est dit programme DC. Il est bien connu qu'en programmation mathématique la résolution de problèmes de minimisation convexe n'est pas en soit un problème difficile, il est alors évident que les problèmes NP-complets ne peuvent se réduire à un programme mathématique de minimisation convexe. En fait bien souvent il s'agit de problèmes de minimisation concave pouvant se reformuler comme des programmes DC.
Les applications de la programmation DC sont nombreuses [36; 52; 92] et concernent tout aussi bien des problèmes tirés de la pratique que des problèmes d'optimisation plus théorique. On peut citer :

1. Méthode de région de confiance par programmation DC [91].
2. Combinaison de DCA et d'algorithmes de points intérieurs pour la résolution de problèmes non linéaires, quadratique non convexe et en zero-un [2].

3. Programmation linéaire en variable mixte (application à l'ordonnancement, logistique, télécommunication) [72; 74; 93].
4. Combinaison de DCA et d'une métaheuristique (Cross-entropy) pour des problèmes de finance, affectation et recherche d'informations [73].
5. Combinaison de DCA avec un algorithme génétique pour des problèmes de fouille de données et en cryptologie [51].
6. Conception d'une chaîne logistique [54; 93].
7. Traitement d'image et vision par ordinateur [74].

Cette liste n'est évidemment pas exhaustive, le succès de la programmation DC tient en fait beaucoup à sa réussite dans la résolution de problèmes de grande taille.

3.2 Notions de base de l'analyse DC

Le lecteur pourra se référer à [5; 36; 83] pour un exposé plus détaillé en particulier la dualité DC.

Adoptons les notations et conventions suivantes :

- Notons $\mathbb{R}^n$ par X,
- Notons par Y le dual de X, qu'on peut identifier à X lui-même,
- Notons $\overline{\mathbb{R}} = \mathbb{R} \bigcup \{-\infty, +\infty\}$,
- $\overline{\mathbb{R}}$ a une structure déduite de celle de $\mathbb{R}$ avec la convention $-\infty - (+\infty) = +\infty$.

Définition 3.2.1.

- *Un segment de ligne entre deux points x,y* $\in \mathbb{R}^n$ *est un ensemble défini par :*

$$[x, y] = \{\lambda x + (1 - \lambda)y, \lambda \in [0, 1]\}.$$

- *Un sous-ensemble C de* $\mathbb{R}^n$ *est convexe, si pour tout x,y de C, le segment [x,y] est inclus dans C, i.e.*

$$\forall x, y \in C, \forall \lambda \in [0, 1] : (\lambda x + (1 - \lambda)y) \in C.$$

Définition 3.2.2. *On dit qu'une fonction f :C*$\subset \mathbb{R}^n \to \mathbb{R}$ *est coercive, si*

$$\lim_{\|x\| \to +\infty} f(x) = +\infty$$

.

Définition 3.2.3. *Soit C un convexe de $\mathbb{R}^n$ et f une fonction de C dans $\mathbb{R}$, f est dite convexe sur C si elle vérifie :*

$$\forall x, y \in C, \forall \lambda \in [0,1] : f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y)).$$

Si l'inégalité est stricte pour tout $\lambda \in]0,1[$ et tout x,$y \in C$ avec $x \neq y$, alors f est dite strictement convexe.

Définition 3.2.4.

- *Soit un ensemble $C \subset X$. On appelle fonction indicatrice de C, la fonction notée χ_C, défini par :*

$$\chi_C(x) = \begin{cases} 0 & si \quad x \in C \\ +\infty & sinon \end{cases} \tag{3.1}$$

.

- *La fonction indicatrice χ_C est convexe si et seulement si C est un ensemble convexe.*
- *Soit $C \subset X$ un ensemble convexe. La fonction $f : C \to \mathbb{R}$ est convexe si et seulement si la fonction $f + \chi_C$ est convexe sur X à valeurs dans $\mathbb{R} \cup \{+\infty\}$.*

Définition 3.2.5. ***(Fonction fortement convexe)*** *La fonction $f : C \to \mathbb{R}$ est dite fortement convexe de module ρ sur l'ensemble convexe C (autrement dit ρ-convexe) lorsqu'il existe $\rho > 0$ tel que pour tout x et $y \in C$ et tout $\lambda \in [0,1]$, on ait :*

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y) - \lambda(1-\lambda)\frac{\rho}{2}\|x-y\|^2.$$

Le module de forte convexité de f sur C, noté $\rho(f, C)$ est défini par :

$$\rho(f, C) = \sup\{\rho > 0 : f - \frac{\rho}{2}\|.\|^2 \text{ est convexe sur } C\}.$$

On dit que f est fortement convexe sur C si $\rho(f, C) > 0$.

Remarquons que toute fonction fortement convexe est strictement convexe, et toute fonction strictement convexe est convexe.

Définition 3.2.6. *Soit $S \subseteq \mathbb{R}^n$ un ensemble convexe, et $f : S \to \overline{\mathbb{R}}$:*

- *le* domaine effectif *de f noté dom(f) est défini par :*

$$dom(f) = \{x \in S : f(x) < +\infty\}$$

.

- *on note epi(f) l'*épigraphe *de f :*

$$epi(f) = \{(x, \alpha) \in S \times \mathbb{R} : f(x) \leq \alpha\}$$

.

- *On dit que f est* propre *si dom(f)$\neq 0$ et $f(x) > -\infty$, $x \in S$.*
- *f est dite* semi-continue inférieurement *(s.c.i) en un point $x \in S$, si :*

$$\liminf_{y \to x} f(y) \geq f(x)$$

.

Définition 3.2.7.
- *Soit f une fonction convexe et propre sur X, $y^0 \in Y$ est dit* sous-gradient *de f au point $x^0 \in dom(f)$, si :*

$$< y^0, x - x^0 > + f(x^0) \leq f(x), \quad \forall x \in X.$$

- *L'ensemble de tous les sous-gradients de f en x^0 est dit* sous-différentiel *de f au point x^0 et est noté par $\partial f(x^0)$.*

Définition 3.2.8.
- *Etant donné un nombre positif $\epsilon > 0$, un élément $y^0 \in Y$ est dit* ϵ-sous-gradient *de f au point x^0, si :*

$$< y^0, x - x^0 > + f(x^0) \leq f(x) + \epsilon, \quad \forall x \in X.$$

- *L'ensemble de tous les ϵ-sous-gradients de f au point x^0 est dit* ϵ-sous-différentiel *de f au point x^0 et est noté $\partial_\epsilon f(x^0)$.*

On note $\Gamma_0(X)$, l'ensemble des fonctions convexes, semi-continues inférieurement et propres sur X.

Définition 3.2.9. *Soit f :X$\to$ $\mathbb{R}$ une fonction quelconque, la fonction conjuguée de f, notée f^* est définie sur Y par :*

$$f^*(y) = \sup\{< x, y > - f(x) : x \in X\}.$$

f^* est l'enveloppe supérieure des fonctions affines continues $y \to < x, y > - f(x)$ sur Y.

Proposition 3.2.1. *Si $f \in \Gamma_0(X)$, alors :*

1. *f$\in \Gamma_0(X) \Leftrightarrow f^* \in \Gamma_o(Y)$, dans ce cas on a : $f = f^{**}$,*
2. *$y \in \partial f(x) \Leftrightarrow f(x) + f^*(y) =< x, y >$ et $y \in \partial f(x) \Leftrightarrow x \in \partial f^*(y)$,*
3. *$\partial f(x)$ est une partie convexe fermée,*
4. *Si $\partial f(x) = \{y\}$, alors f est différentiable en x et $\nabla f(x) = y$,*
5. *$f(x^0) = \min\{f(x), \quad x \in X\} \Leftrightarrow 0 \in \partial f(x^0)$.*

Définition 3.2.10.

- *Soit S$\subseteq$X. L'enveloppe convexe de S, noté co(S) est l'ensemble des combinaisons convexes finies d'éléments de S, c'est à dire :*

$$co(S) = \{\sum_{i=1}^{n} \lambda_i x^i / \lambda_i \in \mathbb{R}^+, x^i \in S, \forall i = \overline{1,n} \text{ et } \sum_{i=1}^{n} \lambda_i = 1\}.$$

- *Soit $C \subseteq \mathbb{R}^n$ un ensemble convexe, on définit la variété linéaire engendré par C comme étant l'ensemble :*

$$aff(C) = \{\sum_{i=1}^{n} \lambda_i x^i / \lambda_i \in \mathbb{R}, x^i \in C, \forall i = \overline{1,n} \text{ et } \sum_{i=1}^{n} \lambda_i = 1\}$$

Définition 3.2.11.

- *Une partie convexe C est dite convexe polyédrale si*

$$C = \bigcap_{i=1}^{m} \{x :< a_i, x > -\alpha_i \leq 0\} \text{ où } a_i \in \mathbb{R}^n, \alpha_i \in R, \forall i = 1, ..., m.$$

- *f est dite convexe polyédrale, si :*

$$f(x) = \sup\{< a_i, x > -b_i, i = 1, ..., k\} + \chi_C(x)$$

où :

- *C est une partie convexe polyédrale.*
- *χ_C désigne la fonction indicatrice de C.*

Proposition 3.2.2.

- *Soit f une fonction convexe polyédrale. f est partout finie si et seulement si C=X.*
- *si f est polyédrale alors f^* l'est aussi. De plus si f est partout finie alors :*

$$f(x) = \sup\{< a_i, x > -b_i, i = 1, ..., k\}.$$

$$dom(f^*) = co\{a_i, i = 1, ..., k\}.$$

$$f^*(y) = \min\{\sum_{i=1}^{k} \lambda_i a_i : y = \sum_{i=1}^{k} \lambda_i a_i, \lambda_i \geq 0, \sum_{i=1}^{k}\}$$

Définition 3.2.12. *Soit C un sous-ensemble convexe de $\mathbb{R}^n$. Une fonction $f : C \to \mathbb{R}$ est dite DC sur C, s'il existe deux fonctions convexes $g,h : C \to \mathbb{R}$, telles que f puisse être exprimée sous la forme*

$$f(x) = g(x) - h(x), \quad \forall x \in C. \tag{3.2}$$

Si $C = \mathbb{R}^n$, alors f est simplement dite fonction DC. Toute représentation de la forme (3.2) est dite décomposition DC de f.

Proposition 3.2.3. *Soit f et f_i, $i=1,...,m$ des fonctions DC. Alors, les fonctions suivantes sont aussi DC :*

1. $\sum_{i=1}^{m} \lambda_i f_i(x), \quad \lambda_i \in \mathbb{R}, \quad i = 1, ..., m,$
2. $\max_{i=\overline{1,m}} f_i(x)$, $\min_{i=\overline{1,m}} f_i(x)$,
3. $|f(x)|$, $f^+(x) = \max\{0, f(x)\}$, $f^- = \min\{0, f(x)\}$,
4. $\prod_{i=1}^{m} f_i(x)$.

Un important résultat de caractérisation des fonctions DC tiré de [39].

Définition 3.2.13. *Une fonction DC f est dite localement DC, si pour tout $x_0 \in \mathbb{R}^n$, il existe $\epsilon > 0$ telle que f est DC sur la boule :*

$$B(x_0, \epsilon) = \{x \in \mathbb{R}^n : \|x - x_0\| \leq \epsilon\}$$

Proposition 3.2.4. *Toute fonction localement DC est DC.*

Proposition 3.2.5.

1. *Toute fonction $f : \mathbb{R}^n \to R$ dont les dérivées partielles de second ordre sont partout continues est DC.*
2. *Soit C un sous-ensemble compact et convexe de $\mathbb{R}^n$. Alors, toute fonction continue sur C est limite d'une suite de fonctions DC sur C, au sens de la convergence uniforme ; i.e., pour toute fonction $c : C \to \mathbb{R}$ et pour tout $\epsilon > 0$, il existe une fonction DC $f : C \to \mathbb{R}$ telle que : $|c(x) - f(x)| \leq \epsilon, \forall x \in C$,*
3. *Soit $f : \mathbb{R}^n \to \mathbb{R}$ une fonction DC, et $g : \mathbb{R} \to \mathbb{R}$ une fonction convexe. Alors la composition $(g \circ f)(x) = g(f(x))$ est DC.*

3.3 Optimisation DC

Un programme mathématique dans lequel figure une fonction DC peut être considéré comme un programme DC. Trois formes de problèmes DC sont répertoriés [92] :

$$\sup\{f(x), x \in C\}, \quad \text{f et C sont convexes,} \tag{3.3}$$

$$\inf\{g(x) - h(x), x \in X\}, \quad \text{g et h sont convexes,} \tag{3.4}$$

$$\inf\{g(x) - h(x), x \in C, f_1(x) - f_2(x) \leq 0\}. \tag{3.5}$$

On peut remarquer que le premier problème est un problème de maximisation convexe. Le problème (3.3) peut être ramené sous la forme (3.4) en considérant la fonction $g \equiv \chi_C$ et $h = f$. Quant au problème (3.5) il peut être ramené à la forme (3.4) via la pénalité exacte relative à la contrainte DC $f_1(x) - f_2(x) \leq 0$.
La forme (3.4) communément appelé *programmation DC* est d'un intérêt majeur d'un point de vue tant théorique que pratique. En effet, d'un point de vue théorique, la dualité DC ainsi que les conditions d'optimalité concernent la forme (3.4), et du point de vue pratique bon nombre de problèmes d'optimisation d'actualité se mettent aisément sous cette forme.

3.3.1 Dualité DC

Considérons le problème DC suivant :

$$\alpha = \inf\{g(x) - h(x) : x \in X\} \quad (P_{DC})$$

Où g et h sont deux fonctions convexes sur X.
La dualité DC repose entièrement sur la notion de fonction conjuguée, le problème dual se construit de cette manière :
Rappelons tout d'abord que : $h(x) = \sup\{< x, y > -h^*(y) : \quad y \in Y\}$, en remplaçant dans

(P_{DC}) :

$$\begin{aligned}
\alpha &= \inf\{g(x) - \sup\{< x, y > -h^*(y) : y \in Y\} : x \in X\} \\
&= \inf_{x\in X}\inf_{y\in Y}\{g(x) + \{h^*(y) - < x, y >\}\} \\
&= \inf_{y\in Y}\inf_{x\in X}\{h^*(y) - < x, y > +g(x)\} \\
&= \inf_{y\in Y}\{h^*(y) - \sup_{x\in X}\{< x, y > -g(x)\}\} \\
&= \inf_{y\in Y}\{h^*(y) - g^*(y)\}
\end{aligned}$$

Le problème dual de (P_{DC}) est alors :

$$\alpha = \inf_{y\in Y}\{h^*(y) - g^*(y)\} \quad (D_{DC})$$

Le problème D_{DC} est aussi un programme DC car les conjuguées h^*, g^* sont aussi convexes, de plus les deux problèmes ont la même valeur optimale. La dualité DC construite sur la base de la dualité Fenchelienne est différente de la dualité Lagrangienne, ainsi on ne retrouve pas le théorème de dualité faible.

Théorème 3.3.1.

- *x^* est une solution optimale globale de (P_{DC}) si et seulement si :*

$$\alpha = (g - h)(x^*) \leq (h^* - g^*)(y), \quad \forall y \in Y \tag{3.6}$$

- *y^* est une solution optimale globale de (D_{DC}) si et seulement si :*

$$\alpha = (h^* - g^*)(y^*) \leq (g - h)(x), \quad \forall x \in X \tag{3.7}$$

Théorème 3.3.2. *Soient g et $h \in \Gamma_0(X)$, alors :*

1. $$\inf_{x\in dom(g)}\{g(x) - h(x)\} = \inf_{y\in dom(h^*)}\{h^*(y) - g^*(y)\} \tag{3.8}$$

2. *Si y^0 est un minimum de $h^* - g^*$ sur Y, alors chaque $x^0 \in \partial g^*(y^0)$ est un minimum de g-h sur X.*

3.3.2 Optimalité globale en optimisation DC

Définition 3.3.1. *Un point x^* est dit point critique (point de KKT généralisé) du problème (P_{DC}) si $\partial g(x^*) \cap \partial h(x^*) \neq \varnothing$.*

Le théorème suivant donne une autre caractérisation des optimums globaux par les ϵ-sous-différentiels tirés des deux théorèmes précédents :

Théorème 3.3.3. *(**Optimalité globale DC**) Soit f=g-h où g,h* $\in \Gamma_0(X)$. *Alors* x^* *est un minimum global de g(x)-h(x) sur X si et seulement si,*

$$\partial_\epsilon h(x^*) \subset \partial_\epsilon g(x^*), \quad \forall \epsilon > 0. \tag{3.9}$$

Le critère d'optimalité globale ainsi formulé n'est malheureusement pas vérifiable en pratique, comme on le verra plus loin en programmation DC polyédrale, l'algorithme DCA converge vers un point extrême du polyèdre convexe qui peut être proche de la solution globale sans néanmoins en être une.

3.3.3 Optimalité locale en optimisation DC

Théorème 3.3.4. *(**Condition nécessaire d'optimalité locale**) Si* $x^\bullet$ *est un minimum local de g-h alors*

$$\partial h(x^\bullet) \subset \partial g(x^\bullet) \tag{3.10}$$

Théorème 3.3.5. ***(Condition suffisante d'optimalité locale)*** *Si* $x^\bullet$ *admet un voisinage V tel que :*

$$\partial h(x) \cap \partial g(x^\bullet) \neq \varnothing, \quad \forall x \in V \cap dom(g) \tag{3.11}$$

alors $x^\bullet$ *est un minimum local de g-h.*

Corollaire 3.3.1. *Si* $x \in int(dom(h))$ *vérifie*

$$\partial h(x) \in int(\partial g(x)), \tag{3.12}$$

alors x est un minimum local de g-h.

Corollaire 3.3.2. *Si* $h \in \Gamma_0(X)$ *est convexe polyédrale alors* $\partial h(x) \subset \partial g(x)$ *est une condition nécessaire et suffisante pour que x soit un minimum local de g-h.*

Corollaire 3.3.3. ***(Transport par dualité DC des minima locaux)*** *Supposons que* $x^\bullet \in dom(\partial h)$ *soit un minimum local de g-h. Soient* $y^\bullet \in \partial h(x^\bullet)$ *et* $V_{x^\bullet}$ *un voisinage de* $x^\bullet$ *tel que* $g(x) - h(x) \geq g(x^\bullet) - h(x^\bullet), \quad \forall x \in V_{x^\bullet} \cap dom(g)$. *Si*

$$x^\bullet \in int(dom(g^*)) \text{ et } \partial g^*(y^\bullet) \subset V_{x^\bullet}, \tag{3.13}$$

alors $y^\bullet$ *est un minimum local de* $h^* - g^*$.

3.4 Algorithme DCA

Le DCA (DC Algorithm) est un algorithme du sous-gradient, basé sur la dualité et l'optimalité DC. C'est un algorithme sans calcul de pas, qui se base sur le problème primal et dual, et en ce sens on peut dire que c'est un algorithme primal-dual. En fait on ne résout pas le problème original directement, mais une succession de problèmes convexes construits par l'utilisation de minorantes affines des composantes DC, permettant d'obtenir de bonnes solutions, même si elles ne sont théoriquement que locales.

3.4.1 Schéma de l'algorithme

L'algorithme DCA repose sur la construction de deux suites $\{x^k\}$ et $\{y^k\}$, dont leurs limites respectives x^* et y^* sont candidates à êtres les solutions respectives du primal et du dual. Ces deux suites vérifient les conditions suivantes de la dualité DC :

1. Les suites $\{g(x^k) - h(x^k)\}$ et $\{h^*(y^k) - g^*(y^k)\}$ sont décroissantes,
2. Si $(g-h)(x^{k+1}) = (g-h)(x^k)$ alors l'algorithme s'arrête à la $(k+1)^{eme}$ itération et le point x^k (resp. y^k) est un point critique de g-h (resp. $g^* - h^*$),
3. Sinon toute valeur d'adhérence $x^\bullet$ de $\{x^k\}$ (resp. $y^\bullet$ de $\{y^k\}$) est un point critique de g-h (resp. $h^* - g^*$).

L'algorithme converge donc vers un couple $(x^\bullet, y^\bullet) \in X \times Y$ tel que :

$$x^\bullet \in \partial g^*(y^\bullet) \text{ et } y^\bullet \in \partial h(x^\bullet).$$

L'algorithme DCA peut être donné sous forme simplifié :

Algorithme [DCA-simplifié]

1. Soit x^1 une solution initiale, $k = 1$ et $\epsilon > 0$.
2. A une itération k, x^k étant connu, on détermine $y^k \in \partial h(x^k)$.
3. Trouver $x^{k+1} \in \partial g^*(y^k)$.
4. Si $|(g-h)(x^{k+1}) - (g-h)(x^k)| < \epsilon$ alors l'algorithme s'arrête ; sinon $k = k+1$, aller à l'étape 2.

Calcul des sous-gradients

Comme on peut le voir, l'algorithme DCA se base sur le calcul de sous-gradients des décompositions DC de la fonction primale et duale.
En pratique le calcul du sous-gradient de la fonction h est généralement aisé étant donné qu'il s'agit du problème primal et qu'on dispose de sa forme explicite. Par contre, le calcul du sous-gradient de la conjuguée de g nécessite la résolution d'un problème d'optimisation convexe :

$$\partial g^*(y^k) = argmin\{g(x) - < y^k, x >: \quad x \in X\}. \tag{3.14}$$

Ainsi, à l'étape 3 de l'algorithme DCA, on a :

$$x^{k+1} \in \partial g^*(y^k) \Leftrightarrow x^{k+1} \in argmin\{g(x) - [< y^k, x - x^k > + h(x^k)] : \quad x \in X\}. \tag{3.15}$$

L'algorithme DCA peut être schématisé de la manière suivante :

$$x^k \quad \longrightarrow \quad y^k \in \partial h(x^k) \quad \longrightarrow \quad x^{k+1} \in \partial g^*(y^k) \quad \longrightarrow \quad y^{k+1} \in \partial h(x^{k+1})$$

L'algorithme DCA est donc un algorithme de point fixe des multi-applications ∂g^* et ∂h :

$$x^{k+1} \in (\partial g^* \circ \partial h)(x^k). \tag{3.16}$$

3.4.2 Existence et convergence des suites générées

On dira que l'algorithme DCA est bien défini si on peut construire les deux suites $\{x^k\}$ et $\{y^k\}$ de l'algorithme DCA-simplifié à partir d'un point initial quelconque.

Lemme 3.4.1. *Les suites $\{x^k\}$ et $\{y^k\}$ générés par DCA sont bien définies si et seulement si :*

$$dom(\partial g) \subset dom(\partial h) \text{ et } dom(\partial h^*) \subset dom(\partial g^*). \tag{3.17}$$

Lemme 3.4.2. *Si g-h est coercive alors on a :*

1. *La suite $\{x^k\}_k$ est bornée,*
2. *Si $\{x^k\}_k \subset int(dom(h))$, alors la suite $\{y^k\}_k$ est aussi bornée.*

Par dualité, si $(h^* - g^*)$ est coercive alors on a :

1. *La suite* $\{y^k\}_k$ *est bornée,*
2. *Si* $\{y^k\}_k \subset int(dom(g^*))$*, alors la suite* $\{x^k\}_k$ *est aussi bornée.*

Théorème 3.4.1. ***(Convergence de l'algorithme DCA)*** *On suppose que les suites* $\{x^k\}$ *et* $\{y^k\}$ *sont bien définies.*

1. *Les suites* $\{g(x^k) - h(x^k)\}$ *et* $\{h^*(y^k) - g^*(y^k)\}$ *sont décroissantes et*
   - *(g-h)*(x^{k+1})*=(g-h)*(x^k) *si et seulement si*

     $y^k \in \partial g(x^k) \cap \partial h(x^k), y^k \in \partial g(x^{k+1}) \cap \partial h(x^{k+1})$ *et* $[\rho(g) + \rho(h)]\|x^{k+1} - x^k\| = 0.$

     De plus, si g et h sont strictement convexes sur X, alors $x^k = x^{k+1}$*. Dans ce cas, DCA se termine en un nombre fini d'itérations.* x^k *et* x^{k+1} *sont des points critiques de la fonction g-h.*
   - $(h^* - g^*)(y^{k+1}) = (h^* - g^*)(y^k)$ *si et seulement si*

     $x^{k+1} \in \partial g^*(y^k) \cap \partial h^*(y^k), x^{k+1} \in \partial g^*(y^{k+1}) \cap \partial h^*(y^{k+1})$ *et* $[\rho(g^*) + \rho(h^*)]\|y^{k+1} - y^k\| = 0.$

     De plus, si g^* *et* h^* *sont strictement convexes sur Y, alors* $y^k = y^{k+1}$*. Dans ce cas, DCA se termine en un nombre fini d'itérations.* y^k *et* y^{k+1} *sont des points critiques de la fonction* $h^* - g^*$*.*
2. *Si* $\rho(g) + \rho(h) > 0$ *(resp.* $\rho(g^*) + \rho(h^*) > 0$*), alors la suite* $\{\|x^{k+1} - x^k\|^2\}$ *(resp.* $\{\|y^{k+1} - y^k\|^2\}$*) converge.*
3. *Si la valeur optimale du problème* P_{DC} *est finie et si les suites* $\{x^k\}$ *et* $\{y^k\}$ *sont bornées, alors toute valeur d'adhérence* x^* *(resp.* y^**) de* $\{x^k\}$ *(resp.* $\{y^k\}$*) est un point critique de g-h (resp.* $h^* - g^*$*).*

3.4.3 Optimisation DC polyédrale

Considérons le programme DC (P_{DC}). L'optimisation DC polyédrale concerne des programmes DC où l'une des composantes DC (g ou h) est polyédrale, c'est une classe particulière de problèmes d'optimisation DC qui possèdent des propriétés intéressantes notamment celle concernant sa convergence.

Restreignons-nous au cas où h est polyédrale (par dualité si g est polyédrale le même raisonnement peut être fait sur le problème dual) :

$$h(x) = \max_{x \in X} \{< a_i, x > -b_i : \quad i = 1, ..., m\}$$

La première remarque qui peut être faite concerne le calcul des sous-gradients $y^k \in \partial h(x^k)$, il est clair que $y^k \in \{a_i : \quad i = 1, ..., m\}$. De plus par la nature des suites générées par DCA (décroissance), la suite $\{y^k\}$ sera finie car l'algorithme ne revient pas à un point déjà exploré et de ce fait à son sous-gradient. On obtient alors un algorithme convergeant au bout d'au plus m itérations.

Remarque 3.4.1. ***(Interprétation de DCA)*** *Remarquons qu'à chaque itération k de l'algorithme DCA, la partie concave de la fonction objectif est remplacé par sa majorante affine au point x^k sur tout l'ensemble convexe, le problème obtenu devient alors convexe.*

3.5 DCA pour la résolution de programmes linéaires à variables mixtes

Nous présentons dans ce chapitre l'algorithme DCA combiné à une procédure de séparation et évaluation pour la résolution des programmes linéaires à variables mixtes 0-1 [72].

3.5.1 Reformulation

Soit à résoudre le programme linéaire suivant :

$$(PL01) \begin{cases} \min c^T x + d^T y \\ A(\frac{x}{y}) \leq b, \\ A_{eq}(\frac{x}{y}) = b_{eq}, \\ x \in \{0,1\}^n, y \in \mathbb{R}^p_+. \end{cases} \tag{3.18}$$

où :

- A est une matrice $m \times (n + p)$, $b \in \mathbb{R}^m$,

- A_{eq} une matrice $l \times (n+p)$, $b_{eq} \in \mathbb{R}^l$,
- n le nombre de variables binaires et p le nombre de variables continues,
- $c \in \mathbb{R}^n$ et $d \in \mathbb{R}^p$.

On définit les deux ensembles S (ensemble des solutions réalisables) et K (ensemble des solutions de la relaxation linéaire de S) :

$$S = \{(x,y) \in \{0,1\}^n \times \mathbb{R}^p_+ : \quad A(\frac{x}{y}) \leq b, \quad A_{eq}(\frac{x}{y}) = b_{eq}\},$$

$$K = \{(x,y) \in [0,1]^n \times \mathbb{R}^p_+ : \quad A(\frac{x}{y}) \leq b, \quad A_{eq}(\frac{x}{y}) = b_{eq}\}.$$

La difficulté dans la résolution du problème (PL01) réside dans les contraintes de binarité. L'idée de base de la reformulation des contraintes de binarité consiste à trouver une fonction q :[0,1]→ $\mathbb{R}$ atteignant son minimum aux points 0 et 1. En voici deux exemples :

1. $q_1(x) = \min(x, 1-x), \quad x \in [0,1]$,
   - q_1 atteint son minimum sur [0,1] aux points 0 et 1,
   - q_1 est concave,
   - $q_1 \geq 0, \quad \forall x \in [0,1]$,
   - q_1 est partout differentiable sauf en $\frac{1}{2}$.
2. $q_2(x) = x - x^2, \quad x \in [0,1]$,
   - q_2 atteint son minimum sur [0,1] aux points 0 et 1,
   - q_2 est concave,
   - $q_2(x) \geq 0, \quad \forall x \in [0,1]$,
   - q_2 est partout differentiable sur [0,1].

Il s'ensuit la reformulation suivante des contraintes de binarité :

$$x \in \{0,1\}^n \Leftrightarrow p(x) = \sum_{i=1}^{n} q(x_i) = 0, x \in [0,1]^n. \tag{3.19}$$

Etant donné que $p(x) \geq 0$ sur $[0,1]^n$, (3.19) est équivalent à :

$$x \in \{0,1\}^n \Leftrightarrow p(x) = \sum_{i=1}^{n} q(x_i) \leq 0, x \in [0,1]^n. \tag{3.20}$$

où q est l'une des fonctions q_1 ou q_2. Grâce à l'équation (3.20), le problème PL01 peut s'écrire sous la forme d'un programme non linéaire :

$$NLP \begin{cases} \min c^T x + d^T y \\ A(\frac{x}{y}) \leq b, \\ A_{eq}(\frac{x}{y}) = b_{eq}, \\ p(x) \leq 0, \\ x \in [0,1]^n, y \in \mathbb{R}^p_+. \end{cases} \tag{3.21}$$

Théorème 3.5.1. ***(Pénalité exacte)****[53] Soit K un polytope de $\mathbb{R}^n$. Soit f une fonction concave finie sur K et soit p une fonction concave finie et non négative sur K. Alors il existe un nombre fini $\tau_0 \geq 0$ tel que les deux problèmes suivants sont équivalents pour tout $t \geq \tau_0$ (au sens où ils ont le même ensemble de solutions optimales) :*

$$\alpha(t) = \inf\{f(x) + tp(x), x \in K\},$$

$$\alpha = \inf\{f(x) : x \in K, p(x) \leq 0\}.$$

De plus, si l'ensemble de points extrêmes V(K) de K est contenu dans $\{x \in K, p(x) \leq 0\}$, alors $\tau_0 = 0$, sinon $\tau_0 = \min\{\frac{f(x)-\alpha(0)}{\xi}\}$, où $\xi = \min\{p(x) : x \in V(K), p(x) > 0\} > 0$.

Puis le théorème de pénalité exacte nous permet de transformer le problème (3.21) en un problème pénalisé :

$$PLP \begin{cases} \min c^T x + d^T y + tp(x) \\ A(\frac{x}{y}) \leq b, \\ A_{eq}(\frac{x}{y}) = b_{eq}, \\ x \in [0,1]^n, y \in \mathbb{R}^p_+. \end{cases} \tag{3.22}$$

Avec $p(.)$ définie par (3.19). Il est évident que le problème pénalisé est un problème de minimisation concave. En particulier, pour q_2 on obtient un programme quadratique concave. Le caractère NP-complet du problème de départ dû à sa nature combinatoire ne disparaît pas pour autant et s'exprime dans la difficulté inhérente au problème d'optimisation globale tout aussi NP-complet.

Notons par K l'ensemble des solutions de la relaxation linéaire du problème (3.18) :

$$K = \{A(\frac{x}{y}) \leq b, \quad A_{eq}(\frac{x}{y}) = b_{eq}, \quad x \in [0,1]^n, \quad y \in \mathbb{R}^p_+\}. \tag{3.23}$$

Le problème (3.22) peut s'écrire sous forme d'un programme DC :

$$\inf\{g(x,y) - h(x,y) : (x,y) \in \mathbb{R}^{n+p}\}. \tag{3.24}$$

où :

- $g(x,y) = \chi_K(x,y) + c^T x + d^T y$, où $\chi_K(x,y)$ est la fonction indicatrice de l'ensemble convexe K, définie par (3.1).
- $h(x,y) = -tp(x)$.

Comme K est convexe, sa fonction indicatrice est elle aussi convexe, g est alors somme de fonctions convexes, donc convexe. D'autre part, la fonction $p(x)$ étant concave, alors h sera convexe. Le problème (3.24) est bien un programme DC.

3.5.2 DCA appliqué au problème (3.24)

Suivant le schéma général de DCA, étant donnée une solution courante $z^k = (x^k, y^k)$ il nous faut calculer

$$u^k \in \partial h(z^k), \quad z^{k+1} \in \partial g^*(u^k).$$

Calcul du sous-gradient

Pour une solution donnée $z = (x,y)$, le calcul du sous-gradient [72]

$$u = (\mu, \nu) \in \partial h(x,y)$$

ne dépend que des variables $x_i, \quad i = \overline{1,n}$, il est donné par :

1. pour $p(x) = \sum_{i=1}^{n} min(x_i, 1 - x_i)$:
   - $$\nu = 0, \quad \mu_i = \begin{cases} -t & \text{si } x_i \leq \frac{1}{2}, \\ t & \text{sinon.} \end{cases} \tag{3.25}$$
2. pour $p(x) = \sum_{i=1}^{n} x_i - x_i^2$:
   - $$\nu = 0, \quad \mu = e - 2x, \text{où } e = (1, ..., 1)^T \in \mathbb{R}^n. \tag{3.26}$$

DCA appliqué au problème PLP peut se décrire ainsi :

Algorithme 5[72]

1. Soit $z^0 = (x^0, y^0) \in K$ un point initial, $k = 0$,
2. Calculer $u^k = (\mu^k, \nu^k) \in \partial h(x^k, y^k)$ grace à l'une des formules 3.25,3.26,
3. Résoudre le programme linéaire : $z^{k+1} = argmin\{c^T x + d^T y - < u^k, z >, z \in K\}$,
4. Si $|h(x^{k+1}) - h(x^k)| < \epsilon$ alors arrêter
   sinon aller à l'étape 2.

Théorème 3.5.2. ***(Convergence de l'algorithme 5)****[72]*

1. *L'algorithme 5 génère une suite $\{z^k\}$ contenue dans l'ensemble des points extrêmes de K telle que la suite $\{g(z^k) - h(z^k)\}$ soit décroissante.*
2. *Si à l'itération r, $r \geq 1$, le point (x^r, y^r) satisfait $x^r \in \{0,1\}^n$, alors (x^k, y^k) satisfait aussi $x^k \in \{0,1\}$ pour tout $k \geq r$.*
3. *La suite $\{z^k\}$ converge vers une solution $z^* \in V(K)$ après un nombre fini d'itérations et z^* est un point critique du problème pénalisé 3.22. De plus, si $x_i^* \neq 0.5, \quad \forall i = 1, ..., n$, alors z^* est une solution locale du problème 3.22.*

L'algorithme 5 ne trouve pas forcément la solution optimale du problème PL01 mais tente de trouver parmi tous les points extrêmes celui qui offre le meilleur compromis entre l'objectif principal et l'objectif induit par la pénalité (qui rend compte de la partie fractionnaire de la solution).

Afin de trouver la solution optimale, des techniques issues de l'optimisation combinatoire ont été combinées à DCA, les plus connues sont la méthode de séparation et évaluation [72] et les techniques de coupes [71].

3.6 Globalisation de DCA par séparation et évaluation

La méthode de séparation et évaluation [72] est basée sur une recherche arborescente d'une solution optimale, connue pour résoudre des problèmes combinatoires NP-difficiles.

Considérons le problème suivant :

$$(P)\begin{cases}\min f(x),\\ x \in X.\end{cases} \tag{3.27}$$

où X est l'ensemble des solutions réalisables. On s'intéresse à l'optimum global $x^* \in X$: $f(x^*) \leq f(x), \forall x \in X$, où l'énumération des éléments de X est impossible en raison de sa cardinalité. La méthode de séparation et évaluation cherche alors à éliminer de l'espace de recherche des sous-ensembles qui ne peuvent fournir une solution optimale.

3.6.1 Principe de la méthode

Le principe de cette approche consiste à séparer le problème en sous-problèmes en imposant à cet ensemble des contraintes supplémentaires, les ensembles des solutions réalisables deviennent plus restreints. Puis des tests appliqués aux sous-problèmes générés permettent de supprimer ceux qui ne peuvent donner l'optimum global.
Cette méthode de décomposition du problème initial peut être schématisé sous forme d'une arborescence, illustré dans la figure 3.1 (d'où le nom de recherche arborescente) :

- Chaque sous-problème créé au cours de l'exploration est symbolisé par un sommet, la racine étant le problème originel,
- Les branches de l'arborescence schématisent le processus de séparation.

Plus précisément, la méthode de séparation et évaluation repose sur trois principes :

- Principe de séparation,
- Principe d'évaluation,
- Stratégie de parcours.

1. **Principe d'évaluation :**

   Le principe d'évaluation permet de diminuer l'espace de recherche. L'objectif est d'essayer d'évaluer l'intérêt de l'exploration d'un sous-problème de l'arborescence.

   Pour ce faire, deux bornes sont utilisées (afin d'éliminer des branches de l'arborescence) :

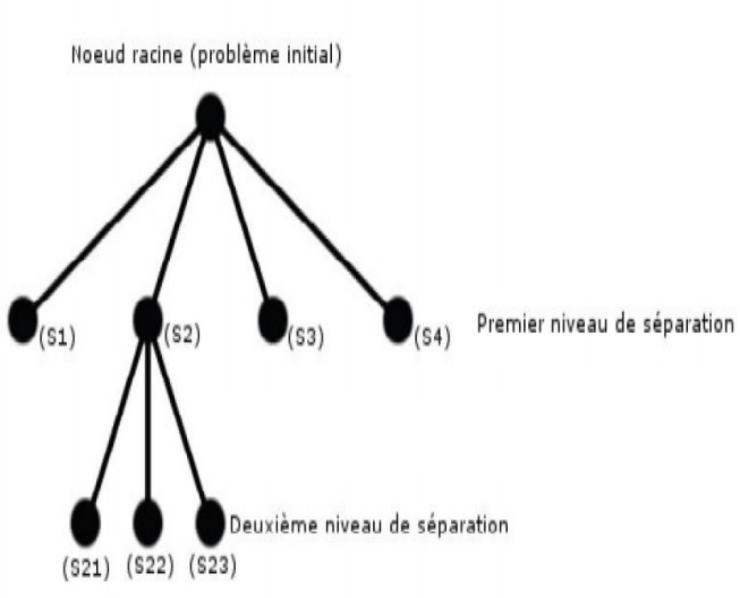

FIGURE 3.1: Schéma d'un processus de séparation-évaluation

- Une borne inférieure de la fonction objectif,
- Une borne supérieure trouvée au préalable.

La connaissance d'une borne inférieure et supérieure permettent d'éliminer certains sous-problèmes non candidats à fournir une solution meilleure. Ainsi si pour un sous-problème donné, sa borne inférieure est supérieure ou égale à sa borne supérieure, alors il est inutile d'explorer le sous-problème.

2. **Principe de séparation :**

   Le principe de séparation englobe l'ensemble des règles qui régissent la séparation d'un problème en sous-problèmes, tout en assurant l'optimalité de la solution trouvée. Pour ce faire, le processus de séparation doit obéir aux règles suivantes :

   - L'ensemble des solutions du problème initial doit être égal à l'union de toutes les solutions des sous-problèmes générés (aucune solution optimale n'est écartée).
   - La difficulté des sous-problèmes doit être inférieure à celui du problème initial.
   - Pour un sous-problème qu'on ne peut séparer, on doit connaître sa meilleure solution, ou une solution meilleure que toutes celles du sous-problème ou bien encore que le sous-problème ne permet pas de fournir une solution admissible.

3.6.2 Schéma général d'un algorithme de séparation et évaluation (SE)

Soit à résoudre le problème (3.27), Notons dans ce qui suit par :

1. S_i^k un sous-ensemble de S^k,
2. $\mu(S)$ une borne inférieure du problème $\min\{f(x), x \in S\}$,
3. γ^k une borne supérieure du problème $\min\{f(x), x \in S^k\}$.

Définition 3.6.1. *Soit $M \subset \mathbb{R}^n$ et soit I un ensemble fini d'indices. L'ensemble $\{M_i : i \in I\}$ est appelé partition de M si :*

- $M = \bigcup_{i \in I} M_i$,
- $M_i \cap M_j = \partial M_i \cap \partial M_j$, $\forall i, j \in I : i \neq j$,
- $M_i, \quad i \in I$, *sont des compacts de* M

où ∂M_i est la frontière de M_i relative à M.

Algorithme 6 : Schéma d'algorithme SE :

Initialisation :

1. $S^1 = X$,
2. Calculer une borne inférieure $\mu(S^1)$ de f sur S^1, $\mu(S^1) = \inf_{x \in S^1} f(x)$.
3. Calculer une borne supérieure $\gamma^1 = f(\xi^1)$ de f sur $Q^1 = Q(S^1) \subset S^1$, où Q^1 est l'ensemble des solutions réalisables de 3.27 incluses dans S^1 ($\gamma_1 = +\infty$ si $Q(S^1) = \emptyset$),
4. $\mu^1 = \mu(S^1)$, $\mathcal{R}^1 = \{S^1\}$, k=1.

Itération k :

1. Si $\gamma_k = \mu_k$ alors arrêter l'algorithme : ξ^k est la solution optimale, sinon partitioner S^k en r sous-ensembles $S_1^k, ..., S_r^k$,
2. Pour tout i=1,...,r :

   - Calculer $\mu(S_i^k)$: $\mu(S_i^k) \geq \mu_k$

- Determiner $\gamma^{k+1} = f(\xi^{k+1}) = \min\{f(x), \quad x \in Q^{k+1}\}$,
  où $Q^{k+1} = Q^k \cup \{Q(S_i^k), \quad i = 1, ..., r\}$,

3. Soit $\mathcal{R}^{k+1}$, la collection des sous-ensembles générés par séparation jusqu'à l'étape k :

$$\mathcal{R}^{k+1} = \mathcal{R}^k \backslash \{S^k\} \cup \{S_i^k, i = 1, ..., r\},$$

4. supprimer tout sous-ensemble ne pouvant fournir la solution optimale, i.e.,

$$\mathcal{R}^{k+1} = \{S \in \mathcal{R}^k : \mu(S) < \gamma^{k+1}\},$$

5. Si $\mathcal{R}^{k+1} = \emptyset$, alors $\gamma_{k+1} = \mu_{k+1}$, sinon choisir S^{k+1}
6. $k \leftarrow k + 1$, Retourner à l'étape 1 de l'itération k.

Remarque 3.6.1. *L'algorithme SE présenté ci-dessus construit les sous-problèmes à chaque itération en adoptant la stratégie du meilleur d'abord.*

Algorithme 7 : algorithme SE pour (PL01) :

Initialisation :

1. $S^1 = X$,
2. Résoudre la relaxation linéaire de (PL01) pour obtenir $\mu(S^1) = f(z^r)$,
   Si $(z^r \in S)$ alors z^r est la solution optimale de (PL01), sinon $\gamma_1 = +\infty$, $k \leftarrow 0$, $\mu_1 = \mu(S^1)$, $\mathcal{R} = \{S^1\}$.

Itération k :

1. Si $(\gamma_k = \mu_k)$ alors arrêter l'algorithme, sinon aller à l'étape 2.
2. (Choix du sous-problème à explorer)
   (2.1) Choisir $S^k \in \mathcal{R} : \mu(S^k) = \min\{\mu(S), S \in \mathcal{R}\} = f(z^k) = f(x^k, y^k)$,
   (2.2) Aller à l'étape 3.
3. (Séparation du sous-problème)
   (3.1) Soit $x_l^k \notin \{0, 1\}$, séparer S^k en deux sous-ensembles :

$$S_0^k = \{z \in S^k, x_l = 0\}, \quad S_1^k = \{z \in S^k, x_l = 1\}$$

(3.2) Aller à l'étape 4.

4. (Evaluation)

   (4.1) Calculer $\mu(S_0^k) = \mu_0^k = f(z_0^k)$ et $\mu(S_1^k) = \mu_1^k = f(z_1^k)$,

   (4.2) Si $(z_0^k \in S$ et $\mu_0^k < \gamma_k)$ alors $\gamma_k = \mu_0^k$, $\xi = z_0^k$,

   Si $(z_1^k \in S$ et $\mu_1^k < \gamma_k)$ alors $\gamma_k = \mu_1^k$, $\xi = z_1^k$

5. $\mathcal{R} = \mathcal{R} \backslash \{S^k\} \cup \{S_0^k, S_1^k\}$,

   $\mathcal{R} = \{S \in \mathcal{R} : \mu(S) < \gamma_k\}$,

   Si $\mathcal{R} = \emptyset$ alors arrêter l'algorithme et ξ est la solution optimale, sinon $k \leftarrow k + 1$, retourner à l'étape 1 de l'itération k.

Le succès de l'algorithme de séparation et d'évaluation repose sur le choix fait au cours des étapes 2) et 3), dans lesquelles on doit faire un choix pouvant conduire à une nouvelle solution réalisable autant qu'à l'exploration de sous-problème ne contenant pas de solution.

On trouve dans la littérature diverses stratégies d'exploration et de séparation.

3.6.3 Stratégie d'exploration

Lors de la phase 2 de l'algorithme 7, on doit choisir un sous-problème à explorer parmi plusieurs obtenus lors de l'étape précédente. Il existe plusieurs façons de choisir un tel sous-problème [1], elles ne sont que des choix heuristiques visant à faire converger l'algorithme le plus rapidement possible.

Parmi les plus connues, on trouve l'exploration en profondeur d'abord, en largeur d'abord et l'exploration meilleure d'abord. Des stratégies incorporant le calcul parallèle sont apparues [7; 81], dans lesquelles plusieurs sous-problèmes sont explorés simultanément.

1. **Exploration en profondeur d'abord** : avec stratégie, on sélectionne le dernier sous-problème engendré, elle a le mérite de nécessiter que peu d'espace mémoire et très utilisée en début d'exploration afin de trouver une première solution rapidement, néanmoins si le sous-problème choisi ne peut conduire à une solution réalisable alors l'algorithme reste piégé jusqu'à l'exploration complète de toutes les alternatives

possibles restantes (s'il reste m variables non encore fixées, au pire cas l'algorithme devra exécuter 2^m itérations avant de sortir de cette branche).

2. **Exploration largeur d'abord** : cette stratégie explore tous les sous-problèmes engendrés, les sous-problème obtenus à chaque étape sont donc tous sur le même niveau. c'est une stratégie gourmande en temps et en mémoire, elle peut permettre d'améliorer la borne inférieure et de proposer une liste de sous-problème variés pour une stratégie d'exploration alternative.
3. **Exploration meilleure d'abord** : la stratégie meilleure d'abord est la plus utilisée, les sous-problèmes sont valués afin d'en choisir le plus prometteur à donner une solution optimale. La valuation la plus couramment utilisée est la valeur de la fonction objectif, mais l'on peut considérer d'autre types de valuation plus spécifique au problème traité.

3.6.4 Stratégie de séparation

La stratégie de selection [1] de la variable fractionnaire est l'autre paramètre de l'algorithme de séparation et d'évaluation qu'il convient de bien choisir afin de s'assurer une convergence rapide, là encore plusieurs règles heuristiques existent, dont la plupart se base sur une valuation des variables candidates. Nous présentons les plus connues :
Soit $F = \{i \in \{1, .., n\} : 0 < x_i < 1\}$.

1. Variable de plus grande partie fractionnaire : $x_{i^*} = \max\{x_i : i \in F\}$.
2. Variable dont la partie fractionnaire est la plus proche de $\frac{1}{2}$: $x_{i^*} = \min\{|x_i - 0.5| : i \in F\}$.
3. Règle de selection dure : le principe de cette règle proposée dans la version 7.5 de CPLEX est de fixer chacune des variables fractionnaires à 0 puis à 1 et de résoudre le programme linéaire ainsi généré, puis de sélectionner la variable permettant la plus forte diminution de la fonction objectif. Elle est à juste titre dite de règle dure puisqu'elle implique la résolution de plusieurs programmes linéaires, ce qui entraîne des temps de calcul prohibitif.
4. Règle de selection basé sur la notion de pseudo-coût : c'est une règle plus complexe, puisqu'elle tient compte de l'historique des séparations effectuées. Soit Q_i^- et Q_i^+

(resp. $C_{Q_i^-}$ et $C_{Q_i^+}$) les deux sous-problèmes (resp. les valeurs de la fonction objectif) engendrés par la fixation de la variable x_i à 0 et à 1 à partir du problème initiale Q, $f_i^+ = \lceil x_i \rceil - x_i$ et $f_i^- = x_i - \lfloor x_i \rfloor$, $\Delta_i^- = C_Q - C_{Q_i^-}$ et $\Delta_i^+ = C_Q - C_{Q_i^+}$, et soit $\varsigma_i - = \frac{\Delta_i^-}{f_i^-}$ et $\varsigma_i^+ = \frac{\Delta_i^+}{f_i^+}$, et soit σ_i^+ la somme des quantités ς_i^+ sur tout les problèmes explorés jusque là et η_i^+ le nombre de ces sous-problèmes. Le pseudo-coût est défini alors par la quantité $\Psi_i^+ = \frac{f_i^+ \sigma_i^+}{\eta_i^+}$. En utilisant la valuation $v_i = \mu \min(\Psi_i^+, \Psi_i^-) + (1-\mu)\min(\Psi_i^-, \Psi_i^+)$, où μ est un paramètre fixant l'importance accordée aux deux valeurs possible 0 et 1, on choisit alors la variable ayant la plus grande valuation.

5. Règle de choix aléatoire : on choisit une variable fractionnaire de manière aléatoire.

3.6.5 Combinaison DCA-SE

Dans l'**algorithme SE pour PL01**, les bornes supérieures sont trouvées en résolvant une relaxation linéaire d'un certain sous-problème, de ce fait cette borne supérieure coincide avec la borne inférieure de ce même sous-problème. Afin d'améliorer la recherche de borne supérieure, on rajoute à l'algorithme précédent la résolution des sous-problèmes par DCA.

Le calcul des bornes inférieures se fait par la résolution de la relaxation linéaire, quand à la borne supérieure on la calcule à présent en résolvant le programme DC relatif au sous-problème engendré par la procédure de séparation.

Algorithme 8 : DCA-SE pour PL01 :

Initialisation :

1. $S^1 = X$,
2. Résoudre la relaxation linéaire de PL01 pour obtenir $\mu(S^1) = f(z^r)$,
   Si ($z^r \in S$) alors z^r est la solution optimale de PL01,
   sinon
   - calculer z_{DCA}, la solution donnée par DCA avec z^r comme point initial, poser $\gamma_{DCA} = f(z_{DCA})$,

- si($z_{DCA} \in S$) alors $\gamma_k = \gamma_{DCA}$, sinon $\gamma_1 = +\infty$, $\mu_1 = \mu(S^1)$, $\mathcal{R} = \{S^1\}$, $k \leftarrow 1$.

Itération k :

1. Si ($\gamma_k = \mu_k$) alors arrêter l'algorithme, sinon aller à l'étape 2.
2. (Choix du sous-problème à explorer)
   (2.1) Choisir $S^k \in \mathcal{R} : \mu(S^k) = \min\{\mu(S), S \in \mathcal{R}\} = f(z^k) = f(x^k, y^k)$,
   (2.2) Aller à l'étape 3.
3. (Séparation du sous-problème)
   (3.1) Si $x_l^k \notin \{0, 1\}$, séparer S^k en deux sous-ensembles :
   $$S_0^k = \{z \in S^k, x_l = 0\}, \quad S_1^k = \{z \in S^k, x_l = 1\}$$
   (3.2) Aller à l'étape 4.
4. (Evaluation)
   (4.1) Calculer $\mu(S_0^k) = \mu_0^k = f(z_0^k)$ et $\mu(S_1^k) = \mu_1^k = f(z_1^k)$,
   (4.2) Calculer à partir de z_0^k (resp. z_1^k) $\gamma_{DCA,0}^k = f(z_{DCA,0}^k)$ (resp. $\gamma_{DCA,1}^k = f(z_{DCA,1}^k)$),
   (4.3) Si ($z_{DCA,0}^k \in S$ et $\gamma_{DCA,0}^k < \gamma_k$) alors $\gamma = \gamma_{DCA,0}^k$, $\xi = Z_{DCA,0}^k$,
   Si ($z_{DCA,1}^k \in S$ et $\gamma_{DCA,1}^k < \gamma_k$) alors $\gamma_k = \gamma_{DCA,1}^k$, $\xi = Z_{DCA,1}^k$,
   (4.4) Aller à l'étape 5.
5. $\mathcal{R} = \mathcal{R} \backslash \{S^k\} \cup \{S_0^k, S_1^k\}$,
   $\mathcal{R} = \{S \in \mathcal{R} : \mu(S) < \gamma_k\}$,
   Si $\mathcal{R} = \emptyset$ alors arrêter l'algorithme et ξ est la solution optimale, sinon $k \leftarrow k + 1$, retourner à l'étape 1 de l'itération k.

Remarque 3.6.2. *L'algorithme 8 diffère peu d'un algorithme basé uniquement sur la relaxation linéaire, l'amélioration apportée consiste à calculer une solution DCA à chaque itération. Ces calculs supplémentaires augmentent la durée d'exécution d'une itération, mais peuvent permettre d'obtenir une solution réalisable. Notons aussi que le point initial utilisé est la solution courante de la relaxation linéaire, or le succès de DCA dépend comme pour tout algorithme de son point initial. L'algorithme 8 traite de la programmation linéaire à variables zero-un au sens général, or il n'est pas évident que le point initial utilisé soit le plus indiqué pour une application particulière.*

Chapitre 4

Application de la programmation DC à la résolution du 1D-BPP

4.1 Introduction

Rappelons qu'il s'agit de ranger une liste d'objets $I = \{a_1, ..., a_n\}$ dans des bins de même taille.
Afin de simplifier le programme linéaire, nous déterminons une borne supérieure grâce à l'algorithme du Best Fit Decreasing, puis nous assignons les grands objets dont la taille est supérieure à la moitié de la capacité des bins à des bins différents, puis nous générons le programme linéaire en excluant les rangements impossibles, par exemple si un grand objet est rangé dans un bin donné et qu'un autre ne peut être rangé dans l'espace restant alors la variable correspondante est fixé à 0 et éliminé du problème.

Avant de lancer l'algorithme de séparation et d'évaluation, un premier prétraitement est effectué grâce à la procédure **MTRP** afin de réduire la taille de l'instance, puis on calcule avec l'algorithme de Best fit Decreasing le nombre de bins suffisants pour ranger le restant des objets afin de générer le programme linéaire lui correspondant en veillant à assigner les objets dépassant la moitié de la taille des bins chacun dans une boite différente.

4.2 Application au problème sans conflits

Nous appliquons ici un algorithme différent de l'algorithme DCA-SE présenté plus haut. Dans cet algorithme, on se déplace d'une solution DCA à une autre. L'objectif étant de trouver une solution meilleure que celles obtenues par l'heuristique du Best fit decreasing et celle donné par la procédure 2. Ainsi on ne cherchera que les solutions dont la valeur optimale est égale à celle de la borne inférieure, si aucune solution n'est trouvée on augmentera la borne inférieure d'une unité. Etant donné que l'exploration de branches dont les bornes inférieures sont inférieures à la meilleure solution trouvée est toujours nécessaire dans un algorithme exacte, l'algorithme suivant n'est pas pire que le précédent au pire cas. L'intérêt réside dont le fait de se déplacer d'une solution DCA à une autre puisqu'elle sont en général de meilleures point initiaux que ceux de la relaxation linéaire.

De plus une série de prétraitements est effectués au début puis à chaque itération de l'algorithme. Le premier prétraitement (**procédure 1**) vise à réduire la taille du problème afin d'avoir un programme linéaire plus léger que celui donné au premier chapitre dont lequel un problème à 100 objets nécessite 10100 variables et 200 contraintes, or il est possible de réduire ce problème comme exposé plus bas.
Le second prétraitement (**procédure 2**) s'effectue à chaque itération de l'algorithme afin de fixer certaine variables, on se base pour ce faire sur le critère de dominance de Martello et Toth.

4.2.1 Procédure de réduction

Comme nous l'avons vu au chapitre 2, nous pouvons réduire la taille de l'instance à résoudre en effectuons en appliquons la procédure MTRP. En plus de cela, les objets dont la taille est supérieure à la moitié de la capacité des bins doivent être mis séparément, puis certain cas de figure peuvent être exclues, ainsi si un objet de taille x est supérieure à la taille d'un bin donné alors la variable concerné par un tel rangement doit être mise à zéro.
Etant donné une liste d'objets $\{a_1, a_2, ..., a_n\}$ triés par ordre décroissant de taille. Soient

les éléments suivant :

- $i^* = \max\{i : a_i > C/2\}$.
- m le nombre de bins prévus pour ranger les objets.
- C_j la capacité du j^{eme} bin.
- $I_j = \{i : a_i \leq C_j, i = 1, ..., n\}$.
- $J_i = \{j : y_j \geq a_i, j = 1, ..., m\}$.

Le modèle du problème réduit peut se réécrire ainsi :

$$\min \sum_{j=1}^{m} y_j \tag{4.1}$$

$$\sum_{i \in I_j} a_i x_{ij} - C_j y_j \leq 0, \quad j = 1, ..., m. \tag{4.2}$$

$$\sum_{j \in J_i} x_{ij} = 1, i = 1, ..., n. \tag{4.3}$$

$$x_{ij}, y_j \in \{0, 1\}, i \in I_j, j = \overline{1, m}. \tag{4.4}$$

Procédure 1 (prétraitement et génération du problème) :

- Données : Une liste d'objets $\{a_1, a_2, ..., a_n\}$.
- Résultat : une affectation optimale de certains objets, une liste d'objets restants.

1. Trier par ordre décroissant de taille les objets restants et les renuméroter.

2. Lancer la procédure MTRP et supprimer les objets ainsi rangés du problème initial.

3. Calculer le nombre m de bins nécessaires pour ranger les objets restants par la borne continue L_0.

4. Calculer l'indice i^* du dernier objet, tel que : $a_{i^*} > C/2$.

5. Mettre à jour : $y_j = y_j - a_j, \quad j = 1, ..., i^*$.

6. Générer le problème réduit, avec les formules (4.1),(4.2),(4.3),(4.4).

Remarquons que si $i^* = n$ alors tous les objets seront d'office rangés dans des bins différents, et la solution du problème est alors triviale.

Dès à présent notons par l_{ij}^x (resp. u_{ij}^x) les bornes inférieures (resp. les borne supérieures) des variables x_{ij}.

Au cours de l'exploration de l'ensemble des solutions, nous fixons certaines variables à 1 ou à 0, on peut affiner l'ensemble des solutions en excluant certains cas non réalisables mais que la relaxation linéaire n'exclut pas.

Procédure 2 :

- Donnée : Un problème initial.
- Résultat : Un nouveau problème réduit.

1. Calculer l'espace restant sur chacun des bins, poser $j = 1$.

2. Si $j = m + 1$ alors terminer, sinon aller à 3)

3. Si $a_i > y_j$ alors $u_{ij}^x = 0$, aller en 4).

4. Si $y_j > \sum_{i \in I_j} a_i$ alors $l_{ij}^x = 1, i \in I_j$, sinon aller en 5).

5. S'il existe un objet $a_i = y_j$ alors $l_{ij}^x = 1$ sinon aller à 6).

6. Si on ne peut mettre qu'un seul objet dans le j^{eme} bin, alors mettre le plus grand, aller à 7).

7. $j = j + 1$ et aller à 2).

4.2.2 Choix du point initial pour DCA

Nous avons testé plusieurs choix de points initiaux dont la solution de la relaxation linéaire et des pseudo-solutions basées sur les heuristiques. Nous avons utilisé deux procedures de génération de point initiaux. Nous considérons les objets au préalable rangés par ordre décroissant.

Procédure 3 :

- Entrée : soit le programme linéaire défini par les formules (4.1),(4.2),(4.3),(4.4).
- Sortie : une solution x^{init}

1. Calculer l'espace restant dans les différents bins.

2. Pour i allant de 1 à n faire

2.1 S'il existe un bin pouvant recevoir l'objet i alors :

2.1.1 Soit j l'indice du premier bin pouvant recevoir l'objet i.

2.1.2 Fixer la variable x_{ij}^{init} à 1.

2.1.3 Mettre à jour l'espace restant $y_j = y_j - a_i$

Procédure 4 :

– Entrée : soit le programme linéaire défini par les formules (4.1),(4.2),(4.3),(4.4).

– Sortie : une solution x^{init}.

1. Pour j allant de 1 à m faire

1.1 Résoudre de manière exacte le problème de sac à dos avec comme objectif d'occuper le plus grand espace possible, et soit I l'ensemble des objets constituant la solution.

1.2 fixer à 1 les variables $x_{ij}^{init}, i \in I$.

1.3 fixer à 0 les variable $x_{ik}^{init}, i \in Ik \in \{1, ..., m\}, k \neq j$

La solution obtenue par la **procédure 3** est celle du first fit decreasing, et la deuxième **procédure 4** résout une succession de problème de somme de sous-ensemble. Nous avons constaté que la **procédure 4** donne des solutions assurant une convergence plus rapide de l'algorithme de branch and bound et à l'avantage de faire figurer l'espace inutilisé dans les dernier bins.

Remarquons aussi que si le nombre m est inférieur strictement au nombre de bins utilisés par le best fit ou par l'heuristique basé sur le problème de sac à dos alors il y'aura nécessairement au moins un objet non rangé, dans ce cas la solution est non réalisable mais reste tout de même un point initial pour le DCA.

4.2.3 Algorithme SE et DCA pour le problème (4.1)-(4.4)

L'algorithme suivant a pour but de trouver une solution réalisable au problème défini par les équations (4.1)-(4.4). Etant donné que les solutions DCA sont meilleures que celles

de la relaxation linéaire, on se basera entièrement sur ces solutions.

Algorithme 9 : algorithme SE pour 4.1-4.4 :

Initialisation :

1. $S^1 = X$, $k = 1$.
2. Résoudre la relaxation linéaire de PL01 pour obtenir $\mu(S^1) = f(x_{LP})$ si le problème est réalisable, sinon arrêter l'algorithme,
   Si (x_{LP} est réalisable), alors x_{LP} est la solution optimale de PL01, sinon aller en 3)
3. Lancer DCA à partir du point initial de la procédure 3 pour obtenir x^1_{DCA}, si la solution est réalisable alors arrêter l'algorithme sinon aller à 4).
4. Lancer DCA à partir du point initial de la procédure 4 pour obtenir x^2_{DCA}, si la solution est réalisable alors arrêter l'algorithme sinon poser $x^k = x^2_{DCA}$ et aller à 5).

   Itération k :
5. Choisir le meilleur sous-problème S^k à explorer, aller à l'étape suivante.
6. Choisir une règle de séparation et séparer le problème en S^k_0 et S^k_1. aller à l'étape suivante.
7. Appliquer la procédure 3 sur S^k_0 et S^k_1, aller à l'étape suivante.
8. Lancer DCA à partir de la solution x^k sur les deux sous-problèmes pour obtenir x^k_0 et x^k_1.
9. Si x^k_0 ou x^k_1 sont réalisables, alors arrêter l'algorithme, sinon aller à l'étape suivante.
10. Mettre à jour l'ensemble des sous-problèmes $\mathfrak{R} = (\mathfrak{R} - S^k) \bigcup \{S^k_0, S^k_1\}$, k=k+1, aller en 12).
11. Si $\mathfrak{R} = \{\}$ alors arrêter l'algorithme, le problème n'a pas de solution, sinon aller à 5).

Remarque 4.2.1. *Comme on peut le voir, l'algorithme 9 génère une séquence de solutions DCA et s'arrête dès qu'il trouve une solution réalisable ou bien montre que l'ensemble des solutions de la relaxation linéaire ne contient pas de solutions réalisables. Cela implique que l'algorithme 9 est complet, de plus le nombre de séparation est borné supérieurement*

par 2^n, où n est le nombre de variables, donc l'algorithme 9 se termine après un nombre fini d'étapes (quoique potentiellement exorbitant). Néanmoins la solution fournie n'est pas forcément optimale, pour remédier à cela nous incorporons l'algorithme 9 dans une procédure (algorithme 10) qui nous permettra de nous assurer de l'optimalité de la solution.

Algorithme 10 : :

1. Appliquer la **procédure 1**.
2. Poser $m = L_1$.
3. Générer le problème réduit, avec m le nombre de bins.
4. Résoudre le problème par **l'algorithme 9**.
5. Si une solution optimale est trouvée alors arrêter l'algorithme, sinon poser $m = m+1$ et aller à 3.

L'algorithme 10 cherche la solution optimale au problème (4.1)-(4.4) en restreignant l'exploration à l'ensemble des solutions le plus restreint possible, cela se justifie par le fait que pour certaines instances les heuristiques utilisées permettent de fournir une solution dont le nombre de bins n'excède pas le nombre optimal de bins plus un, donc appliquer l'algorithme 8 sur ces instances implique que la séquence générée se base sur les solutions de la relaxation linéaire, donc plus mauvaises que celles fournies par DCA, de plus garder les solutions DCA en plus de celle de la relaxation linéaire entraîne un surcoût en termes d'espace mémoire. L'algorithme 10 est une tentative de remédier à ces inconvénients.

Proposition 4.2.1. *La solution fournie par l'algorithme 10 est optimale.*

Démonstration. Deux cas sont à distinguer, soit l'algorithme 9 donne une solution réalisable dont la valeur est égale à la borne inférieure donc forcément optimale, soit l'ensemble est vide, donc qu'il n'existe aucune solution réalisable dont la valeur est égale à la borne inférieure (en particulier la solution optimale), on peut alors incrémenter cette borne d'une unité. En recommençant le même procédé, la solution fournie est à tous les coups optimale. □

4.2.4 Stratégie d'exploration et de séparation pour le problème (4.1)-(4.4)

Stratégie d'exploration

A l'étape 5) de l'algorithme précédent, le choix du sous-problème de meilleure borne inférieure n'a pas de sens vu que tout sous-problème a la même borne inférieure, nous pouvons définir d'autre critère pour le choix du meilleure sous-problème, comme par exemple le sous-problème possédant le plus grand nombre de variables fixées ou bien celui ayant le plus d'objets rangés, ou même celui où figure le plus petit nombre d'objet fractionné. Il faudra s'assurer tout de même que la stratégie d'exploration assure une convergence rapide vers une solution réalisable.

Stratégie de séparation

Les règles de sélections présenté plus haut sont assez générales et peuvent s'appliquer dans tout algorithme de séparation-évaluation pour une programmation à variables binaires. Afin d'améliorer la convergence de l'algorithme il convient de définir des règles de sélection plus efficace tenant compte de la nature du problème en particulier en usant du critère de dominance. Soit x_{ij}^f est une variable fractionnaire.

1. Règle 1 : si en fixant $x_{ij}^f = 1$ on ne peut plus mettre un autre objet dans le bin j alors choisir :
$$x_{ij} = \max\{a_i \wedge l_{ij}^x \neq u_{ij}^x i \in I_j\}$$
2. Règle 2 : parmi toutes les rangements possibles d'objets dans le bin j, ne retenir que les rangements non dominés. On se limite aux ensembles de cardinalités égales à 2 (pour les singletons on utilisera la règle 1). soit E l'ensemble des objets pouvant être mis dans le bin j, et v_{KP} la taille maximale des objets pouvant être mis dans le sac-à-dos (solution du problème de sac-à-dos) et P l'ensemble des paires constituants les solutions du problème de sac-à-dos :
$$E = \{a_i, i \in I_j\}$$
$$P = \{(a_i, a_k) \in E^2 : a_i + a_k = v_{KP}\}$$

Alors la séparation s'effectuera non plus suivant les variables mais suivant les paires de l'ensemble P, pour une paire d'objet (a_{i_1}, a_{i_2}) trois alternatives s'offre à nous, soit cette paire est rangé dans la boite j soit un des objets ne l'est pas :

(a) $l^x_{i_1 j} = l^x_{i_2 j} = 1$.

(b) $u^x_{i_1 j} = 0 \vee u^x_{i_2 j} = 0$.

On peut restreindre le nombre d'alternatives à deux sous certaine conditions pour respecter le schéma de séparation de l'algorithme précédent. Ainsi si en supprimant l'objet a_{i_1} on ne peut former une autre paire optimale pour le problème de sac-à-dos avec l'objet a_{i_2} alors dans la seconde alternative on aura forcément $x_{i_2 j} = 0$, si l'objet $a_{i_2} j$ ne figure lui aussi que dans une seule paire optimale alors on pourra rajouter la contrainte $x_{i_1 j} = 0$ dans la seconde alternative. Il est possible néanmoins qu'on ne puisse pas restreindre le nombre d'alternatives à 2, dans ce cas on choisira la variable fractionnaire la plus proche de 0.5 .

4.2.5 Résultats numériques

Nous appliquons l'algorithme ci-dessus aux instances de Falknauer (http ://people.brunel.ac.uk/ m jjb/jeb/info.html) à 120 et 250 (les benchmarks sont notés u_120 et u_250) objets qui ont été résolus à l'optimum par Valério [96], on arrête le déroulement de l'algorithme après 60 secondes. L'algorithme a été exécute sur un PC portable avec une architecture 64 bits,un microprocesseur i5 et une RAM de 6GO.

On note par :

- n_{obj} : le nombre d'objet à ranger.
- n_{var} : le nombre de variables.
- n_{cont} : le nombre de contraintes.
- iter : le nombre d'itération de l'algorithme.
- n_{sp} : le nombre de sous-problèmes générés.
- t_{exe} : le temps d'exécution de l'algorithme en secondes.
- t_{valrio} : temps d'exécution de l'algorithme exacte.
- bsup : nombre de bins dans la meilleure solution trouvée.
- opt : nombre bins de la solution optimale.

n^o	n_{var}	n_{cont}	iter	n_{sp}	t_{exe}	t_{valrio}	bsup	opt
0	2111	102	30	56	6.80	3.35	48	48
1	1582	90	1	0	0.09	3.74	49	49
2	1881	98	1	0	0.26	4.18	46	46
3	1521	86	19	32	3.98	4.07	49	49
4	1755	94	1	0	0.09	3.35	50	50
5	1540	85	1	0	0.15	4.29	48	48
6	1381	83	1	0	0.14	4.23	48	48
7	1674	93	19	34	4.52	3.02	49	49
8	1498	87	17	27	3.48	5.66	50	50
9	1799	97	19	31	4.32	3.74	46	46
10	1297	78	1	0	0.07	4.95	52	52
11	1499	84	14	24	2.49	3.02	49	49
12	1355	82	1	0	0.16	9.73	48	48
13	1238	79	1	0	0.08	3.07	49	49
14	1344	83	1	0	0.08	4.12	50	50
15	1688	91	1	0	0.21	3.68	48	48
16	1159	75	1	0	0.07	3.41	52	52
17	1304	79	21	38	4.45	2.91	52	52
18	1546	90	1	0	0.18	4.72	49	49
19	1669	91	1	0	0.21	5.16	49	49

TABLE 4.1: Résultats de l'application de l'algorithme 10 sur les instances u_120 (120 objets).

n^o	n_{var}	n_{cont}	iter	n_{sp}	t_{exe}	t_{valrio}	bsup	opt
0	6060	178	18	32	15.73	3.46	99	99
1	5551	167	1	0	0.79	5.49	100	100
2	5150	160	1	0	0.62	5.38	102	102
3	4844	162	1	0	0.53	4.34	100	100
4	4787	159	36	66	18.99	6.43	101	101
5	5165	159	1	0	0.80	6.43	101	101
6	3826	143	1	0	0.38	5.05	102	102
7	6050	173	110	111	60.83	6.65	104	103
8	4166	147	324	109	60.4	10.60	106	105
9	5456	165	1	0	0.64	6.15	101	101
10	4349	148	1	0	0.63	4.67	105	105
11	5221	163	1	0	0.66	6.04	101	101
12	4422	151	128	34	60.62	10.50	106	105
13	6545	178	20	26	66.08	6.04	104	103
14	5953	171	1	0	0.83	4.73	100	100
15	4298	149	20	34	14.65	9.12	105	105
16	6727	184	1	0	0.92	3.85	97	97
17	6404	183	1	0	1.04	4.62	100	100
18	5380	167	13	22	8.96	5.44	100	100
19	5169	165	1	0	0.93	4.67	102	102

TABLE 4.2: Résultats de l'application de l'algorithme 10 sur les instances u_250 (250 objets).

4.3 Application au cas avec conflit

4.3.1 Formulation réduite du problème

Soit :

- $O = \{a_1, a_2, ..., a_n\}$ la taille des n objets à ranger.
- C la capacité des bins.
- m le nombre de bins prévue pour ranger les objets (calculé par l'heuristique du FFD-C).
- $L = \{(l,k) \in \{1,...,n\}^2 :$ l'objet l est en conflit avec l'objet k$\}$.
- $G = \{i \in \{1,...,n\} : a_i > \frac{C}{2}\}$.
- $I = \{i \in \{1,...,n\} : a_i \leq \frac{C}{2}\}$.
- $i^* = \max\{i \in \{1,...,n\} : a_i > \frac{C}{2}\}$.
- $J_i = \{k \in \{1,...,n\} : a_l + a_k \leq C \wedge (l,k) \notin L\}, \quad i = 1,...,i^*$.

L est l'ensemble des couples d'objets en conflit, et J_i est l'ensemble des objets pouvant être rangé avec l'objet i. On peut commencer par ranger les objets de l'ensemble G, puis de ne considérer dans le modèle que les objets qui ne sont pas en conflit avec ces objets et qui peuvent être rangés sans dépasser la capacité du bin, défini par l'ensemble J_i. Le modèle devient :

$$\min \sum_{j=1}^{m} y_j \tag{4.5}$$

$$\sum_{i \in J_j} a_i x_{ij} - (C - a_j) y_j \leq 0, \quad j = 1,...,i^*. \tag{4.6}$$

$$\sum_{i \in I} a_i x_{ij} - C y_j \leq 0, \quad j = i^* + 1,...,m. \tag{4.7}$$

$$\sum_{j \in J_i} x_{ij} = 1, \forall i. \tag{4.8}$$

$$x_{ij}, y_j \in \{0,1\}, j = \overline{1,m}, i \in I_j. \tag{4.9}$$

4.3.2 Point initial du DCA

Nous avons choisi comme point initial, la solution partielle donnée par l'heuristique $\mathfrak{FFDC}$ pour un nombre m de bins (on range les objets si possible sinon il y'aura des objets non rangés).

4.3.3 Résultats numériques

On applique pour ce problème la même démarche que pour le problème précédent, on génère le problème avec un nombre m égale à la borne inférieure de la relaxation continue, puis si le problème est non réalisable, on incrémente la borne d'une unité. N'ayant pas trouvé de Benchmark pour le 1D-BPPC, on génèrera les conflits sur des instances tirées de celle de Falknauer. On considérera des instances de tailles de 20, 40, 60 (5 instances pour chacune des tailles) et un pourcentage de conflits de 20, 50, 70, 90. Il y'aura en tout 60 instances. L'algorithme a été exécute sur un PC portable avec une architecture 64 bits,un microprocesseur i5 et une RAM de 6GO, pour une durée maximum de 200 secondes. Les programme linéaires sont résolus par GLPK. Les instances traité ont été résolu à l'optimum.

Soit :

- p : la probabilité de conflit entre deux objets.
- n_{taille} : la taille de l'instance.
- n_{var} : le nombre de variables.
- n_{cont} : le nombre de contraintes.
- n_{iter} : le nombre d'itérations de l'algorithme.
- n_bin : nombre de bin de la solution optimale.
- t_{exe} : le temps d'exécution de l'algorithme.
- i : un indice pour chaque instance.
- - : indique que l'instance n'a pas été résolue.

(i,p,n_{taille})	n_{var}	n_{cont}	n_{iter}	n_{bin}	t_{exe}
(1,0.2,20)	252	481	2	11	0.08
(2,0.2,20)	252	418	15	10	0.60
(3,0.2,20)	252	443	14	9	0.66
(4,0.2,20)	336	679	2	14	0.08
(5,0.2,20)	315	575	1	13	0.06
(6,0.2,40)	984	3887	20	19	5.19
(7,0.2,40)	1107	3719	20	21	11.89
(8,0.2,40)	943	3797	19	19	5.75
(9,0.2,40)	1066	3610	853	25	60.423
(10,0.2,40)	1066	3791	7	23	1.18
(11,0.2,60)	2013	10926	95	28	93.60
(12,0.2,60)	2318	11811	254	32	75.97
(13,0.2,60)	2318	13336	41	32	89.64
(14,0.2,60)	2440	13789	219	36	63.61
(15,0.2,60)	2501	14627	143	35	68.19

TABLE 4.3: Résultats numériques pour un taux de conflit de 20%

(i,p,n_{taille})	n_{var}	n_{cont}	n_{iter}	n_{bin}	t_{exe}
(16,0.5,20)	273	1133	2	11	0.13
(17,0.5,20)	336	1548	15	10	1.25
(18,0.5,20)	252	980	14	9	0.96
(19,0.5,20)	294	1192	2	14	0.14
(20,0.5,20)	357	1488	1	13	0.12
(21,0.5,40)	1148	10225	20	19	12.10
(22,0.5,40)	1148	10391	20	21	29
(23,0.5,40)	1271	11132	19	19	14.77
(24,0.5,40)	1271	12183	251	25	61.71
(25,0.5,40)	1353	12691	7	23	3.9
(26,0.5,60)	2562	36465	53	29	197.58
(27,0.5,60)	2806	40061	37	32	119.26
(28,0.5,60)	2745	35551	49	33	162.41
(29,0.5,60)	2989	41296	28	36	79.32
(30,0.5,60)	2967	41167	29	35	88.23

TABLE 4.4: Résultats numériques pour un taux de conflit de 50%

(i,p,n_{taille})	n_{var}	n_{cont}	n_{iter}	n_{bin}	t_{exe}
(31,0.7,20)	378	2032	2	11	0.18
(32,0.7,20)	336	1984	15	10	1.28
(33,0.7,20)	336	1936	14	9	1.41
(34,0.7,20)	378	2301	2	14	0.17
(35,0.7,20)	378	2355	1	13	0.14
(36,0.7,40)	1271	16339	20	19	19.36
(37,0.7,40)	1312	16744	20	21	44.40
(38,0.7,40)	1353	17453	19	19	23.26
(39,0.7,40)	1394	17129	149	25	63.01
(40,0.7,40)	1394	17539	7	23	5.82
(41,0.7,60)	2684	52239	-	-	-
(42,0.7,60)	2867	57914	-	-	-
(43,0.7,60)	2989	59925	-	-	-
(44,0.7,60)	2989	57092	-	-	-
(45,0.7,60)	3172	63226	-	-	-

TABLE 4.5: Résultats numériques pour un taux de conflit de 70%

(i,p,n_{taille})	n_{var}	n_{cont}	n_{iter}	n_{bin}	t_{exe}
(31,0.9,20)	420	3195	2	11	0.27
(32,0.9,20)	378	2809	15	10	1.53
(33,0.9,20)	378	2864	14	9	1.69
(34,0.9,20)	378	2971	2	14	0.20
(35,0.9,20)	399	2962	1	13	0.16
(36,0.9,40)	1640	26634	20	19	33.21
(37,0.9,40)	1599	26245	25	22	71.14
(38,0.9,40)	1640	26796	19	19	38.07
(39,0.9,40)	1640	27076	61	25	66.75
(40,0.9,40)	1558	25347	7	23	10.6
(41,0.9,60)	3599	90263	-	-	-
(42,0.9,60)	3599	90384	-	-	-
(43,0.9,60)	3599	92331	-	-	-
(44,0.9,60)	3538	89203	-	-	-
(45,0.9,60)	3416	85571	-	-	-

TABLE 4.6: Résultats numériques pour un taux de conflit de 90%

4.3.4 Discussion des résultats

Les résultats obtenus pour le premier problème (1D-BPP) montre que l'approche par programmation DC est intéressante, vu les temps de résolutions de quelques secondes, mise à part 4 instances pour lesquelles l'algorithme n'a pas convergé dans le temps imparti, cela est du au choix de la stratégie d'exploration et de séparation.
Par contre pour le 1D-BPPC, l'algorithme ne converge pas pour des instances de grandes tailles (nombre de contraintes élevées > 50000), par contre les stratégies de séparation dans ce cas n'ont que peu d'importance, vu que l'élagage de sous-problèmes est plus efficace à cause du fait que le problème est fortement contraint.
On peut conclure enfin que l'approche par séparation-évaluation combiné à DCA bute sur les mêmes difficultés pour ce problème qu'avec un algorithme sans DCA, tout de même l'algorithme DCA améliore considérablement la qualité des solutions partielles par rapport à la relaxation linéaire, au sens du nombre de variables fractionnaires.

Conclusion générale et Perspectives

Dans ce travail, nous avons appliqué la programmation DC avec une procédure de globalisation afin de fournir un algorithme exact pour résoudre le problème de bin packing à une dimension avec et sans conflits.

L'approche par programmation DC transforme le problème posé initialement comme étant un problème d'optimisation discrète en un problème d'optimisation globale continue.

Nous avons appliqué l'algorithme ainsi élaboré à des instances tests qui montrent les possibilités mais aussi les difficultés liées à cette approche, le choix du point initial pour DCA ainsi que les stratégies de séparation et d'exploration pour le 1D-BPP sont les paramètres qui déterminent le succès de l'algorithme, par contre pour le 1D-BPPC, la grande taille des programmes linéaires est l'obstacle principal pour ce problème.

Des améliorations possibles peuvent être apportées notamment pour le 1D-BPPC. Adapter la procédure **MTRP** pour la version avec conflits pourrait aider à réduire la grande taille des programmes linéaires induits pour de forts taux de conflits, de plus essayer d'adapter des heuristiques du 1D-BPP ayant montré leur efficacité pour le 1D-BPPC est aussi intéressant. Quand à l'approche de programmation DC et de globalisation : explorer d'autres types d'algorithmes exacts comme le Branch and Price et les méthodes de coupes, et essayer d'y intégrer la programmation DC, développer des stratégies de branchements et d'explorations plus adaptées pour le problème traité sont des voies possibles d'amélioration de l'algorithme de base.

Bibliographie

[1] T. Achterberg, T. Koch, A. Martin ; Branching Rules revisited ; Operations Research Letters 33 (2005) 42 - 54.

[2] F.B. Akoa ; Approches de points intérieurs et de la programmation DC en optimisation non convexe ; Thèse de doctorat à l'institut national des sciences appliquées de Rouen (2005).

[3] G. Belov, G. Scheithauer ; A Branch-and-Cut-and-Price Algorithm for One-Dimensional Stock Cutting and Two-Dimensional Two-Stage Cutting ; Technical Report MATH-NM-03 - revised on October 02, (2003).

[4] J.O. Berkey, P.Y. Wang ; Two-dimensional finite bin packing algorithms ; journal of operational research society ; 38 :423 - 429, (1987).

[5] J.M. Borwein, A.S. Lewis ; Convex analysis and nonlinear optimization ; Springer (2006).

[6] J.M. Bourjolly, V. Rebetz ; An anlysis of lower bounds procedures for the bin packing problem ; Europrean journal of operational research ; Volume 32, Issue 3 (2005) 395 - 405.

[7] M.E. Bouzgarrou, Parallélisation de la méthode du "Branch and Cut" pour résoudre le problème du voyageur de commerce ; Thèse de doctorat Institut National Polytechnique de Grenoble (1998).

[8] E.K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, R. Qu ; A survey of hyper-heuristics ; Computer Science Technical Report No. NOTTCS-TR-SUB - 0906241418 - 2747 (2009).

[9] P.M. Castro, J.F. Oliveira ; Scheduling inspired models for two-dimensional packing problems ; European Journal of Operational Research 215 (2011) 45 - 56.

[10] F.T.S. Chan, K.C. Au, L.Y. Chan, T.L. Lau ; Using genetic algorithms to solve quality related bin packing problem ; Robotics and computer integrated manufacturing (2005).

[11] L.M.A. Chan, D. Simchi-Levi, J. Bramel ; Worst-case analyses, linear programming and the bin packing problem ; Mathematical Programming (1996).

[12] F.K.R. Chung, M.R. Garey, D.S. Johnson ; On packing two-dimensional bins, SIAM J. Algebraic Discrete Meth ; 3 (1982) 66 - 76.

[13] F. Clautiaux, M. Dell'Amico, M. Lori, A. Khanafer ; Lower and upper bounds for the bin packing problem with fragile objects ; Discrete applied mathematics, (2012).

[14] F. Clautiaux, J. Carlier, A. Moukrim ; A new exact method for the two-dimensional orthogonal packing problem ; European Journal of Operational Research 183 (2007) 1196 - 1211.

[15] F. Clautiaux ; Bornes inférieures et méthodes exactes pour le problème de bin packing en deux dimensions avec orientation fixe ; Thèse de doctorat à l'université de Compiègne (2005).

[16] E.G. Coffman Jr., M.R. Garey, D.S Johnson ; Approximation algorithms for bin packing - an updated survey ; Algorithms design for computer system design pages 49 - 106, Spring-Verlag, New York, (1984).

[17] E.G. Coffman Jr., D.S. Johnson, R.E. Tarjan ; performance bounds for level-oriented two-dimensional packing algorithms ; SIAM Journal Comput. 9(1980) 801 - 826.

[18] E.G. Coffman, J. Csirik ; A classification scheme for bin packing theory ; Acta Cybernetica 18 (2007) 47 - 60.

[19] J. Csirik, J.B.G. Frenk, G. Galambos, A.H.G. Rinooy Kan ; Probabilistic analysis of algorithms for dual bin packing problems ; Journal of algorithms (1991).

[20] J. Csirik, J.B.G. Frenk, M. Labbé ; two-dimensional rectangle packing : online methods and results ; Discrete Applied Mathematics 45 (1993) 197 - 204.

[21] H. Dyckhoff ; A typology of cutting and packing problems ; European Journal of Operational Research, 44 : 145 - 159 (1990).

[22] J. El Hayek ; Le problème de bin packing en deux dimensions, le cas non orienté : résolution approchée et bornes inférieures ; Thèse de doctorat à l'université de Compiègne, (2006).

[23] J. El Hayek, A. Moukrim, S. Negre ; New resolution algorithm and pretreatments for the two-dimensional bin packing problem ; Computers & Operations Research 35 (2008) 3184 - 3201.

[24] L. Epstein ; Two-dimensional online bin packing with rotation ; Theoretical Computer Science 411 (2010) 2899 - 2911.

[25] B. Escoffier ; Approximation polynomiale de problèmes d'optimisation : aspects structurels et opérationnels ; Thèse de doctorat université Paris IX (2005).

[26] S. Fekete, J. Schepers ; An exact algorithm for higher dimensional orthogonal packing ; Revised for operations research (2003).

[27] S. Fekete, J. Schepers ; New classes of fast lower bounds for bin packing problems ; Mathematical Programming, 91 : 11 - 31 (2001).

[28] K. Fleszar, K.S. Hindi ; New heuristics for one-dimensional bin packing ; computers & operations research (2000).

[29] C.A. Floudas, C.E. Gounaris ; A review of recent advances in global optimisation ; J Glob Optim (2009) 45 :3 - 38.

[30] J.B. Frenk, G.G. Galambos ; Hybrid next-fit algorithm for the two dimensional rectangle bin packing problem ; Computing 39 (1987) 201 - 217.

[31] F. Furini, E. Malaguti, R.M. Duran, A. Persiani, P. Toth ; A column generation heuristic for the two-dimensional tw-staged guillotine cutting stock problem with multiple stock size ; European Journal of operational research (2011).

[32] M.R. Garey, R.L. Graham, D.S. Johnson, A.C. Yao ; Resource constrained scheduling as generalized bin packing ; journal of combinatorial theory (A) 21, 257 - 298 (1976).

[33] M. Haouari, M. Serairi ; Heuristics for the variable sized bin packing problem ; Computers & Operations Research 36 (2009) 2877 - 2884.

[34] M. Hifi ; Exact algorithms for the guillotine strip cutting/packing problem ; Computers Ops Res. Vol. 25, No. 11, pp. 925 - 940, (1998).

[35] V. Hemmelmayr, V. Schmid, C. Blum ; Variable neighbourhood search for the variable sized bin packing problem ; Computers & Operations Research 39 (2012) 1097 - 1108.

[36] E. Hopper, B. Turton ; Application of genetic algorithms to packing problems -A review- ; Conference on soft computing in engeneering design and manufacturing (1997).

[37] E. Hopper, B. Turton ; A genetic algorithm for a 2D industrial packing problem ; Computers & Industrial Engineering 37 (1999) 375 - 378.

[38] E. Hopper, B. Turton ; A review of the application of meta-heuristic algorithms to 2D strip packing problems ; Artificial Intelligence Review 16 : 257 - 300, (2001).

[39] R. Horst, N.V. Thoai ; DC programming : Overview ; journal of optimization theory and applications : Vol. 103, No. 1, pp. 1 - 43, october 1999.

[40] J. Kang, S. Park ; Algorithms for the variable sized bin packing problem ; European Journal of Operational Research 147 (2003) 365 - 372.

[41] W. Kern, X. Oiu ; Note on non-uniform bin packing games ; Discrete Applied Mathematics.

[42] M. Kenmochi, T. Imamichi, K. Nonobe, M. Yagiura, H. Nagamochi ; Exact algorithms for the two-dimensional strip packing problem with and without rotations ; European Journal of Operational Research 198 (2009) 73 - 83.

[43] A. Khanafer, F. Clautiaux, E. Talbi ; Tree-decomposition based heuristics for the two-dimensional bin packing problem with conflicts ; Computers & Operations Research 39 (2012) 54 - 63.

[44] N.G. Kinnersley, M.A. Langston ; Online variable sized bin packing ; Discrete Applied Mathematics 22 (1988/89) 143 - 148 .

[45] R.E. Korf ; A new algorithm for optimal bin packing ; AAAI (2002).

[46] B. Korte, J. Vygen ; Optimisation combinatoire théorie et algorithmes ; Springer-Verlag France (2010).

[47] B. Kroger ; Guillotineable bin packing : A genetic approach ; European Journal of Operational Research 84 (1995) 645 - 661.

[48] M. Labbé, G. Laporte, H. Mercure ; Capacitated vehicle routing on trees ; Operations Research, 39(6) : 16 - 22, (1991).

[49] M. Labbé, G. Laporte, S. Martello ; An exact algorithm for the dual bin packing problem ; Operations Research Letters 17 (1995) 9 - 18.

[50] K.K. Lai, J.W.M. Chan ; Developing a simulated annealing algorithm for the cutting stock problem ; Computers ind. Engng Vol. 32, No. 1, pp. 115 - 127, (1997).

[51] M. Le Hoai ; Modélisation et optimisation non convexes basées sur la programmation DC et DCA pour la résolution de certaines classes des problèmes en fouille de données et cryptologie ; Thèse de doctorat de l'université Paul Verlaine-Metz (2007).

[52] H.A. Le Thi, T.P. dinh ; The DC (difference of convex functions) programming and DCA revisited with DC models of real world nonconvex optimization problems ; Annals of Operations Research 133, 23 - 46, (2005).

[53] H.A. Le Thi, T.P. Dinh, H. Van Ngai ; Exact penalty and error bounds in DC programming ; J Glob Optim (2012) 52 :509 - 535.

[54] H.A. Le Thi, T.P. Nguyen, P.D Tao ; A continuous DC programming approach to the strategic supply chain design problem from qualified partner set ; European Journal of Operational Research 183 (2007) 1001 - 1012.

[55] Z. Liu ; Complexity of core allocation for the bin packing game ; Operations Research Letters 37 (2009) 225 - 229.

[56] D.S. Liu, K.C. Tan, S.Y. Huang, C.K. Goh, W.K. Ho ; On solving multiobjective bin packing problems using evolutionary particle swarm optimization ; European Journal of Operational Research 190 (2008) 357 - 382.

[57] A. Lodi, S. Martello, D. Vigo ; Recent advances on two-dimensional bin packing problem ; Discrete Applied Mathematics 123 (2002) 379 - 396.

[58] A. Lodi, S. Martello, D. Vigo ; Models and bounds for two-dimensional level packing problems ; Journal of Combinatorial Optimization, 8, 363 - 379, (2004).

[59] A. Lodi, S. Martello, D. Vigo ; Approximation algorithms for the oriented two dimensional bin packing problem ; Eur. J. Oper. Res. 112 (1999) 158 - 166.

[60] A. Lodi, S. Martello, D. Vigo ; Heuristic and metaheuristic approaches for a class of two dimensional bin packing problems ; INFORMS J.Comput. 11(1999) 345 - 357.

[61] K. Loh, B. Golden, E. Wasil ; Solving the one-dimensional bin packing problem with a weight annealing heuristic ; Computers & Operations Research 35 (2008) 2283 - 2291.

[62] M. Maiza, M.S. Radjef, L. Sais ; Continuous lower bound for the variable sized bin-packing problem ; MOSIM'12 (2012).

[63] M. Maiza, A. Labed, M.S. Radjef ; Efficient algorithms for the offline variable sized bin packing problem ; J Glob Optim (2012).

[64] M. Maiza, M.S. Radjef ; Heuristics for solving the bin packing problem with conflicts ; Applied Mathematical Sciences, Vol. 5, 2011, no. 35, 1739 - 1752.

[65] M. Maiza ; Contributions à la résolution de problèmes de rangement avec incompatibilités dans des boites non identiques ; Thèse de doctorat à l'école nationale supérieure d'informatique (2013).

[66] S. Martello, P. Toth ; Knapsack problems (chapitre 8) ; John Wiley & sons Ltd (1990).

[67] S. Martello, D. Vigo ; Exact solution of the two dimensional finite bin packing problem ; Management Sci., 44 :388 - 399, (1998).

[68] M. Moeini ; La programmation DC et DCA pour l'optimisation de portefeuille ; Thèse de doctorat de l'université Paul Verlaine-Metz (2008).

[69] M. Moeini, H. Le Thi ; A DC programming approach for the constrained two dimensional non guillotine cutting problem ; International Conference on industrial engineering and systems management (2011).

[70] I. Morihara, T. Ibaraki, T. Hasegawa ; Bin packing and multiprocessor scheduling problems with side constraint on job types ; Discrete Applied Mathematics 6 (1983) 173 - 191.

[71] B.M. Ndiaye, Le thi Hoai An, Pham Dinh Tao, Yi Shuai Niu ; DC programming and DCA for Large Scale Two Dimensional Packing Problems ; Springer-Verlag Berlin Heidelberg pp. 321 - 330 (2012).

[72] Q.T. Nguyen ; Approches locales et globales basées sur la programmation DC et DCA pour des problèmes combinatoires en variables mixtes 0-1, Applications à la planification opérationnelle ; Thèse de doctorat de l'université Paul Verlaine-Metz (2010).

[73] D.M. Nguyen ; La programmation DC et la méyhode Cross-Entropy pour certaines classes de problèmes en finance, affectation et recherche d'informations ; Thèse de doctorat de l'institut national des sciences appliquées de Rouen (2012).

[74] T.P. Nguyen ; Techniques d'optimisation en traitement d'image et vision par ordinateur et en transport logistique ; Thèse de doctorat de l'université de Paul-Verlaine-Metz (2007).

[75] Y.S. Niu ; Programmation DC & DCA en optimisation combinatoire et optimisation polynomiale via les techniques de SDP ; Thèse de doctorat de l'institut national des sciences appliquées de Rouen (2010).

[76] J. Nocedal, S.J. Wright ; Numerical optimization ; Springer-Verlag (1999).

[77] F.G. Ortmann, N. Ntene, J.H. Van Vuuren ; New and improved level heuristics for the rectangular strip packing and variable-sized bin packing problems ; European Journal of Operational Research 203 (2010) 306 - 315.

[78] I.H. Osman, G. Laporte ; Metaheuristics : A bibliography ; Annals of Operations Research 63 (1996) 513 - 623.

[79] P.M. Pardalos, O.A. Prokopyev, S. Busygin ; Continuous approches for solving discrete optimization problems ; chapitre 2 du Handbook on Modelling for Discrete Optimization ; Springer (2006).

[80] M. Peeters, Z. Degraeve ; Branch-and-pice algorithms for the dual bin packing and maximum cardinality bin packing problem ; European journal of operational research (2004).

[81] A.T. Philips, J.B. Rosen ; A parallel algorithm for constrained concave quadratic global minimization ; Mathematical Programming 42 (1988) 421-448.

[82] D. Pisinger, M. Sigurd ; The two-dimensional bin packing problem with variable bin sizes and costs ; Discrete Optimization 2 (2005) 154 - 167.

[83] R.T. Rockaffellar ; Convex analysis ; Princeton university press (1972).

[84] A.M. Rubinov, Z.Y. Wu ; Optimality conditions in global optimization and their applications ; Math. Program., Ser. B (2009) 120 :101 - 123.

[85] A. Scholl, R. Klein, C. Jurgens ; BISON : a fast hybrid procedure for exactly solving the one-dimensional bin packing problem ; Computers Ops Res. Vol. 24, No. 7, pp. 627 - 645. 1997.

[86] M. Serairi, M. Haouari ; A computational study of lower bounds for the two dimensional bin packing problem ; Electronic Notes in Discrete Mathematics 36 (2010) 891 - 897.

[87] P. Siarry, Z. Michalewicz ; Advances in metaheuristics for hard optimization ; Springer-Verlag (2008).

[88] D. Simchi-Levi ; New worst-case results for the bin packing problem ; Naval Research Logistics 41(1994), 579 - 585.

[89] A. Shapiro, D. Dentcheva, A. Ruszczynski ; Lectures on stochastic programming ; Society for industrial and applied mathematics and the mathematical programming society (2009).

[90] A. Stawowy ; Evolutionary based heuristic for bin packing problem ; Computers & Industrial Engineering 55 (2008) 465 - 474.

[91] P.D. Tao, L.T.H. An ; A DC optimisation algorithm for solving the trust-region subproblem ; SIAM J. OPTIM. Vol. 8, No. 2, pp. 476 - 505, May (1998).

[92] P.D. Tao, L.T.H. An ; Convex analysis approach to DC programming : Theory, algorithms and applications ; ACTA MATHEMATICA VIETNAMICA Volume 22, Number 1, (1997), pp. 289 - 355.

[93] P.N. Thanh, N. Bostel, O. Péton ; A DC programming heuristic applied to the logistics network design problem ; Int. J. Production Economics 135 (2012) 94 - 105.

[94] D.Q. Tran ; Optimisation non convexe en finance et en gestion de production : modèles et méthodes ; Thèse de doctorat de l'université Paul Verlaine-Metz (2011).

[95] J.M. Valério de Carvalho ; LP models for bin packing and cutting stock problems ; European Journal of Operational Research 141 (2002) 253 - 273.

[96] J.M. Valério de Carvalho ; Exact solution of bin packing using column génération and branch and bound ; Annals of Operations Research 86 (1999) 629 - 659.

[97] R. Vahrenkamp ; Random search in the one-dimensional cutting stock problem ; European Joumal of Operational Research 95 (1996) 191 - 200.

[98] L. Wei, W. Oon, W. Zhu, A. Lim ; A goal-driven approach to the 2D bin packing and variable-sized bin packing problems ; European Journal of Operational Research 224 (2013) 110 - 121.

[99] D.P. Williamson, D.B. Shmoys ; The design of approximation algorithms ; Electronic Web Edition www.designofapproxalgs.com (2010).

[100] X. Yang ; Engineering optimization ; JOHN WILEY & SONS, INC. (2010).

[101] M. Yue ; A simple proof of the inequality for the First fit decreasing bin packing algorithm ; Report No.90665, Research institute for discrete mathematics, University of Bonn, (1990).

[102] G. Zhang, X. Cai, C.K. Wong ; Linear time-approximation algorithms for bin packing ; Operations Research Letters 26 (2000) 217 - 222.

Résumé : Nous traitons dans ce travail le problème de bin packing à une dimension (avec et sans conflits) par une méthode exacte de branch & bound et de programmation DC. Le problème de bin packing à une dimension sans conflits consiste à ranger des objets ayant des tailles données dans un minimum de bins (boites) de capacités limitées. Dans la version avec conflits, on ne peut ranger un objet donné dans un même bin qu'un autre avec lequel il est en conflit, le but reste toujours de ranger tous les objets dans un minimum de bins. Le problème ainsi posé est clairement combinatoire, il est de plus NP-complet. Plusieurs approches existent : heuristiques, métaheuristiques, algorithmes exacts. Notre approche s'inscrit dans cette dernière. Nous formulons tout d'abord le problème sous forme d'un PL01 (formulation de Kantorovitch), puis nous le reformulons sous forme d'un programme quadratique concave et enfin comme programme DC, la solution optimale se trouve alors être le minimum global du programme DC. Pour la résolution du problème, nous utilisons un algorithme de Branch & Bound issu de l'optimisation discrète couplé à l'algorithme DCA issu de la programmation DC.
Mots clés : Bin Packing, Programmation DC, Branch & Bound, Optimisation globale.

Abstract : We treat in this paper the problem of one-dimensional bin packing (with and without conflicts) by an exact method of branch & bound and DC programming. The problem of one-dimensional bin packing without conflict is to store objects with sizes given in a minimum of bins (boxes) of limited capacity. In the version with conflict, we can not store a given object in a single bin another with which it is in conflict, the goal is always to store all the objects in a minimum of bins. This problem is clearly combinatorial, it is more NP-complete. Several approaches exist : heuristics, metaheuristics, exact algorithms. Our approach is an exact algorithm. We first formulate the problem as a PL01 (formulation of Kantorovich), then we reformulate it in form of a concave quadratic program and finally as DC program, the optimal solution is then to be the global minimum of DC program. To solve the problem, we use a Branch & Bound algorithm from discrete optimization coupled with the DCA from the DC programming.
Keywords : Bin Packing, DC Programming, Branch & Bound method, Global Optimization.

ملخص: في هذا العمل نقوم بحل مشكل جمع أشياء في صناديق ذات بعد واحد بخوارزمية دقيقة تعتمد على برمجة الدوال المحدبة و طريقة الفروع المقيدة.
المشكلة هي وضع أشياء ذات أطوال مختلفة في أصغر عدد ممكن من الصناديق ذات طول محدود, في حالة وجود نزاع بين شيئين يستلزم وضعهما في صندوقين مختلفين.
هذا المشكل توفيقي صعب. عدة طرق تقوم بحل هذا المشكل : خوارزمية استدلالية, عالية اللاستدلال و دقيقة. خوارزميتنا تنتمي إلى هذه الأخيرة.
نقوم بصياغة هذا المشكل بالبرمجة الخطية ثنائية المتغير, ثم كبرنامج رباعي مقعر و أخيرا كبرنامج فرق دوال محدبة, فحل المشكل هو الحد الأدنى لذالك البرنامج.
لحل هذا المشكل, نستخدم طريقة الفروع المقيدة و خوارزمية فرق الدوال المحدبة في خوارزمية موحدة.
الكلمات الرئيسية: جمع أشياء في صناديق, برمجة فرق دوال محدبة, طريقة الفروع المقيدة.