Méthodes de résolution hybrides pour les problèmes de type knapsack

Nawal Cherfi

UNIVERSITE PARIS I PANTHEON-SORBONNE

U.F.R 27 - Mathématiques & Informatique

Thèse

présentée pour l'obtention du titre de

Docteur de l'Université Paris I

Spécialité : Informatique

par

Nawal CHERFI

MÉTHODES DE RÉSOLUTION HYBRIDES POUR LES PROBLÈMES DE TYPE KNAPSACK

Soutenue publiquement le 20 Novembre 2008 devant la commission d'examen composée de

Président :	**M. Ali Ridah Mahjoub**	Université Paris 9 Dauphine
Rapporteurs :	**M. Didier El Baz**	LAAS-CNRS, Université de Toulouse
	M. Imed Kacem	Université de Technologie de Troyes
Directeur :	**M. Mhand Hifi**	Université de Picardie Jules Verne
Examinateurs :	**M. Alain chateauneuf**	Université Paris I Panthéon-Sorbonne
	M. Loÿs Thimonier	Université de Picardie Jules Verne
	M. Moussa Elkihel	LAAS-CNRS, Université de Toulouse

Mis en page avec la classe Phdlasl

À mes très chers parents que je chéris par dessus tout.

À mes très chèrs frères et sœurs.

"Il faut savoir s'instruire dans la gaieté. Le savoir triste est un savoir mort. L'intelligence est joie."

Voltaire

Remerciements

Je tiens à exprimer ma gratitude à M. Mhand Hifi, Professeur à l'université de Picardie Jules Verne, pour avoir dirigé et encadré mon travail. Je le remercie pour sa disponibilité, son soutien et ses conseils tout au long de ces années de préparation de mon projet de thèse.

Je remercie également M. Ali Ridha Mahjoub, Professeur à l'université Paris 9-Dauphine, pour avoir accepté de présider ce jury et pour l'intérêt qu'il porte à ce travail.

J'exprime ma gratitude à M. Alain Chateauneuf, Professeur à l'université Paris I Panthéon-Sorbonne pour avoir fait partie de ce jury de thèse en tant qu'examinateur.

Je remercie infiniment M.Imed Kacem, Maître de Conférences/HDR à l'université de Technologie de Troyes et M. Didier El Baz Chargé de Recherche au LAAS-CNRS de Toulouse qui ont bien voulu assurer la tâche de rapporteurs malgré les contraintes particulières de temps et d'espace. Leurs observastions ont contribué à améliorer la qualité de ce manuscrit.

C'est avec un grand plaisir que j'exprime mes remerciements à M. Moussa Elkihel, Maître de Conférences à l'université de Toulouse et M. Loïjs Thimonier, Professeur à l'université de Picardie Jules Verne pour avoir accépté de participer à ce jury de thèse en tant qu'examinateurs.

Je tiens à exprimer ma profonde reconnaissance à M. Ivan Lavallée, Professeur à l'université Paris 8, que j'ai eu le plaisir de cotoyer au début de ma thèse au laboratoire LRIA (Paris 8). Je le remercie pour ses conseils, son aide précieuse et surtout ses qualités humaines et pédagogiques.

Je remercie également mes amies Nguyen Thi Minh Luan et Pham Thuy Lien du laboratoire LRIA ainsi que ma très chère amie Souad Abaajan pour leurs encouragements.

Je remercie par ailleurs tous les membre du CERMSEM, permanents ou thésards (et anciens thésards), en particulier Bernard Kouakou, Tarik Belgacem, Saadi Toufik et autres.

Mes pensées vont également à Marie-Lou Margaria et Marie-Carmen Varela. Je les remercie pour le travail qu'elles font, leur amabilité et leur disponibilité.

Je remercie également tous les membres de ma famille, en particulier mes très chèrs parents et ma chère sœur Malika pour le soutien qu'ils m'ont apporté dans les moments difficiles que j'ai connus durant ces dernières années.

Table des matières

Introduction

De nombreux problèmes réels touchant des domaines variés (industriels, économiques, militaires, etc) peuvent être modélisés comme des problèmes d'optimisation combinatoire. Ces derniers consistent à chercher dans un ensemble discret un sous-ensemble parmi les meilleurs sous ensembles de façon à maximiser ou minimiser un critère (ou pluieurs critères) sous certaines contraintes.

En général, les méthodes de résolution que l'on met en œuvre pour résoudre ce type de problème doivent prendre en considération deux facteurs : la qualité des solutions et le temps de résolution. Bien que ces deux facteurs soient généralement liés, parfois il est nécessaire de faire le choix entre trouver une solution (des solutions) optimale(s) ou de se contenter d'une solution (des solutions) approchée(s). Souvent, ce choix est influencé par la nature du problème traité. Toutefois, l'efficacité d'une méthode de résolution dépend de sa complexité temporaire et spatiale. Si la compléxité temporaire est bornée par une fonction polynomiale d'un paramètre caractérisant la taille du problème alors la méthode est dite efficace ou *polynomiale*. Dans la nomenclature de la complexité algorithmique, les problèmes d'optimisation combinatoire pour lesquels on ne connait pas d'algorithmes de résolution polynomiaux, sont dits *NP-difficiles*. Ces problèmes sont réputés les plus difficiles parmi les problèmes d'optimisation combinatoire.

Les méthodes de résolution des problèmes d'optimisation combinatoire peuvent être classées en deux catégories : les méthodes exactes qui garantissent l'optimalité de la solution et les méthodes approchées qui perdent en optimalité pour gagner en efficacité. Le principe de base d'un algorithme exact consiste en général à énumérer l'ensemble des solutions de l'espace de recherche de façon implicite. Cependant, ce type d'algorithme ne permet de résoudre que des problèmes de taille modérée. Autrement, le temps de calcul risque d'augmenter exponentiellement avec la taille du problème. Les méthodes approchées

constituent une alternative intéressante pour la résolution des problèmes d'optimisation de grande taille si l'optimalité n'est pas primordiale. Parmi ces méthodes nous citons les méthodes gloutonnes, les heuristiques et les métaheuristiques représentées essentiellement par les méthodes de voisinage tels que le recuit simulé, la recherche tabou et les algorithmes évolutifs tels que les algorithmes génétiques.
Notons qu'il existe d'autres méthodes de résolution combinant des algorithmes approchés et des méthodes exactes. Ces méthodes, appelées *méthodes hybrides*, représentent un outil assez puissant pour résoudre les problèmes combinatoires.

Dans cette thèse nous nous intéressons à la conception d'algorithmes approchés et exacts pour la résolution d'une classe de problèmes d'optimisation combinatoire. Nous nous concentrons principalement sur le problème du knapsack multidimensionnel à choix multiple, qui est un programme linéaire en nombres entiers. Bien que les Programmes Linéaires en Nombres Entiers (PLNE) peuvent être extrêmement difficiles, ils sont d'une grande utilité vu leurs nombreux secteurs d'applications : logistique et transport, réseaux de télécommunication ou de distribution d'énergie, etc. Face à des problèmes complexes et fortement combinatoires, les algorithmes issus de la programmation linéaire perdent souvent de leur efficacité. Notre objectif est de contourner l'exhaustivité des procédures de la PLNE pour tenter d'échapper à l'exploration combinatoire importante à laquelle nous serons confrontés.

Cette thèse comprend deux parties principales. La première partie est consacrée aux méthodes de résolution approchée pour le problème du knapsack multidimensionnel à choix multiple. La deuxième partie s'interesse à la résolution exacte du même problème par application d'une méthode de "branch-and-cut".

Plus précisément, dans le premier chapitre, nous décrivons quelques problèmes de type knapsack. Nous présentons un résumé de certaines méthodes de résolution exacte et approchée pour le problème de knapsack classique (unidimensionnel). Certaines de ces méthodes sont à la base des méthodes dérivées pour résoudre un large éventail de problèmes de type knapsack. Ensuite, nous nous consacrons au problème du knapsack multidimensionnel à choix multiple. Par la suite, nous présentons certaines de ses applications et les méthodes de résolution qui lui sont consacrées dans la littérature.

Le deuxième chapitre est consacré aux méthodes de la programmation linéaire et la

programmation linéaire en nombres entiers auxquelles nous ferons appel lors de la résolution du problème du knapsack multidimensionnel à choix multiple. Nous commençons par des définitions et un rappel de l'algorithme du simplexe. Puis, nous décrivons la méthode de génération de colonnes et nous détaillons la méthode de "branch-and-price". Nous présentons ensuite un tour d'horizon sur la méthode des plans de coupe et les concepts de base de la théorie polyédrale. Finalement, nous terminerons ce chapitre par un algorithme simplifié de "Branch-and-Cut".

Le troisième chapitre est consacré à la résolution approchée du problème du knapsack multidimensionnel à choix-multiple. Nous présentons deux méthodes hybrides qui s'appuient principalement sur la génération de colonnes et les méthodes d'énumération implicite. La première approche fait coopérer une méthode d'arrondi et un algorithme exact. Elle est composée de deux phases. Une première phase qui applique, de façon itérative, la génération de colonnes et qui arrondit la solution optimale de la relaxation continue obtenue pour fixer une partie des variables. Une deuxième phase résout de façon exacte la partie restreinte du problème. Cette approche est ensuite généralisée en l'introduisant dans une procédure par séparation et évaluation tronquée. Elle sera appliquée de façon périodique sur un ensemble de nœuds séléctionnés à l'aide de deux paramètres intrinsèques introduits pour créer une sorte de diversification dans l'espace de recherche. La deuxième approche s'appuie sur les méthodes de recherche par voisinage. Elle conjuge les efforts de deux méthodes : (i) la méthode de branchement local et (ii) la méthode de génération de colonnes. La méthode de branchement local est une méthode de recherche par voisinage, proposée par Fischetti et Lodi [25] où les voisinages sont obtenus en introduisant des inégalités linéaires non-valides. Nous utilisons la première heuristique comme boite noire pour résoudre les voisinages. Lorsque cette dernière ne parvient pas à améliorer la solution courante, nous utilisons une technique de diversification ou d'intensification. Nous présentons ensuite, une version augmentée imitant une méthode de Branch and Bound (B&B) mais sans garantir l'optimalité des solutions obtenues. Cette version de l'heuristique peut se résumer par les étapes suivantes : (i) construire une solution initiale, (ii) générer un ensemble de nœuds et, (iii) appliquer la deuxième heuristique à un ensemble de nœuds séléctionnés. Finalement, dans une partie expérimentale, nous évaluons la performances des heuristiques sur un ensemble d'instances de la littérature.

Dans le quatrième chapitre, nous proposons un algorithme de résolution exacte en nous appuyant sur une méthode de branch and cut. Nous commençons par proposer des contraintes valides, puis nous présentons des contraintes de couverture locales et globales particulières. Ces dernières sont ensuite incorporées dans un schéma énumératif. Finalement, nous présentons une étude expérimentale préliminaire de l'algorithme sur deux groupes d'instances : un groupe d'instances composé d'instances principalement ardues et un deuxième groupe d'instances composé d'instances aléatoires de taille modérée.

Nous terminerons ce document par une conclusion générale et certaines perspectives de nos travaux futurs.

Chapitre I

Problèmes de type knapsack

Dans ce chapitre nous présentons les problèmes traités dans cette thèse. Nous commençons par décrire le problème du knapsack ordinaire qui est un problème classique de la recherche opérationnelle. Nous discutons quelques applications de ce dernier. Ensuite, des méthodes de résolution, approchées et exactes, qui lui ont associées. Puis, nous présentons le problème du knapsack multidimentionnel à choix multiple que nous allons étudier dans les chapitres suivants. Enfin, nous abordons les méthodes de résolution de la littérature dédiées à ce problème.

I.1 Introduction

Les *problème du knapsack classique* (sac-à-dos) est un problème de sélection qui consiste à maximiser un critère de qualité sous une contrainte linéaire de capacité de ressource. Il doit son nom à l'analogie qui peut être faite avec le problème qui se pose au randonneur au moment de remplir son knapsack : il lui faut choisir les objets à emporter de façon à avoir un sac le plus *utile* possible, tout en respectant son volume, la question qui se pose est de savoir quels objets il doit mettre dans le sac.
Il existe plusieurs variantes du problème du knapsack. Elles diffèrent dans la distribution des objets et des sacs. Dans le problème du knapsack en variables binaires, chaque objet peut être pris dans le sac au plus une fois. En revanche, dans un problème du knapsack borné, nous avons une quantité bornée de chaque type d'objet. Dans le problème du knapsack à choix multiple, les objets sont à choisir à partir de classes disjointes. S'il existe

plusieurs sacs à remplir simultanément, le problème du knapsack est multidimensionnel. Notons que toutes les variantes de problème du knapsack sont NP-difficiles.
Les problèmes knapsacks ont été intensivement étudiés depuis le travail de Dantzig [18] à la fin des années cinquante. Leur importance dans les applications industrielles ainsi qu'en planification financière ont aussi nécéssité la recheche des méthodes de résolution efficaces. Ces problèmes sont aussi d'une grande importance théorique vu qu'ils interviennent comme sous-problèmes dans plusieurs problèmes en nombres entiers. En effet, le problème du knapsack unidimensionnel intervient, par exemple, comme sous-problème dans le problème de découpe (voir Gilmore et Gomory [28]) lors de la génération d'un pivot du Simplexe. Il intervient également dans la résolution du problème d'affectation généralisée [71].
Par ailleurs, les différentes variantes de problème du knapsack, bien qu'elles semblent très proches, ne font pas du tout appel aux mêmes méthodes de résolution (exactes ou approchées) pour une variante donnée. Dans ce chapitre, nous rappelons quelques méthodes de résolution du knapsack unidimensionnel et nous présentons une de ses variantes que nous avons étudiée ainsi que l'essentiel des approches de résolution qui lui ont été dédiées.

I.2 Problème du knapsack classique (KP)

On considère un ensemble d'objets étiquetés de 1 à n. Chaque objet $j \in \{1, ..., n\}$, dispose d'un poids w_j de valeur entière et d'un profit v_j (réel à priori). On dispose d'un sac dont le contenu ne peut excéder une capacité C entière. On désire le remplir de façon à maximiser le profit total des objets emportés, en respectant la contrainte de capacité. Plus formellement, le problème, noté (KP) peut s'écrire de la façon suivante :

$$(KP)\begin{cases} \max Z(x) = & \sum_{j=1}^{n} v_j x_j \\ \text{s.c.} & \sum_{j=1}^{n} w_j x_j \leq C, \\ & x_j \in \{0,1\},\ j \in \{1,\ldots,n\}, \end{cases}$$

On appelle une instance d'un problème du knapsack, la donnée des n profits v_j pour $j = 1, ..., n$, des n poids w_j pour $j = 1, ..., n$ et de la capacité C.
Le vecteur $x := (x_j,\ 0 \leq j \leq n) \in \{0,1\}^n$ est une solution du problème avec $x_j = 1$, si

l'objet j est emporté dans le sac et $x_j = 0$ sinon. Le problème consiste donc à choisir un sous-ensemble d'objets parmi la collection d'objets initialement prévue en vue de maximiser la fonction objectif :

$$Z(x) = \sum_{j=1}^{n} v_j x_j$$

Nous formulons l'hypothèse suivante :

$$\forall j \in \{1, ..., n\}, w_j \leq C \text{ et } \sum_{j=1}^{n} w_j > C.$$

On dit qu'une solution est réalisable, notée $\bar{x}$, si elle vérifie la contrainte de capacité, c'est-à-dire $\sum_{j=1}^{n} w_j \bar{x_j} \leq C$. La solution est dite optimale, notée x^*, si elle est à la fois réalisable et si elle maximise la somme des valeurs profits des objets mis dans le sac. En d'autres termes, pour toute solution réalisable $\bar{x}$, on a :

$$\sum_{j=1}^{n} v_j \bar{x_j} \leq \sum_{j=1}^{n} v_j x_j^*.$$

On peut illustrer le problème KP dans un exemple pratique très simple. Supposons que n projets s'offrent à un investisseur qui dispose d'un fond de "C euros". Sachant que le profit du $j^{ème}$ projet est p_j, $(j = 1, \ldots, n)$ et qu'investir dans ce projet coûte w_j. Alors, un investissement optimal peut être trouvé en résolvant un problème du knapsack en 0-1.

I.2.1 Méthodes de résolution pour le KP

Il est difficile de développer un algorithme polynomial pour le problème du knapsack (KP) vu que ce dernier appartient à la classe NP-difficile. Malgré cette difficulté, plusieurs instances de grande taille peuvent être résolues en une fraction de seconde. Ce résultat impressionant est la récolte de plusieurs décennies de recherche qui ont exposé les différentes propriétés du problème KP et qui ont rendu sa résolution moins difficile. Nous présentons dans les sections I.2.1.1 et I.2.1.3 quelques méthodes de résolution approchée et exacte pour le KP. Pour plus de détails, nous invitons le lecteur à voir Fayard et Plateau [24], Martello et Toth [55], [56] et Pisinger [67] où diverses approches aussi bien exactes qu'approchées sont présentées.

I.2.1.1 Méthodes approchées

Parmi les méthodes heuristiques utilisées pour résoudre le problème (KP), on peut citer la méthode dite gloutonne. Cette dernière consiste à construire une solution de manière incrémentale, en faisant à chaque pas, le choix qui semble localement le meilleur. Autrement dit, faire un choix localement optimal, dans l'espoir que ce choix mènera à une solution optimale globale.

Comme il existe plusieurs variantes pour ces méthodes gloutonnes, nous présentons ici une des versions. Tout d'abord, les objets sont ordonnés selon l'ordre décroissant du rapport (profit par poids), c'est-à-dire :

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq .... \geq \frac{v_j}{w_j} \geq ... \geq \frac{v_n}{w_n} \tag{I.1}$$

L'algorithme glouton que nous présentons dans la figure I.1 consiste donc à sélectionner à chaque étape un élément selon l'ordre précédemment défini. Si le poids de l'élément séléctionné ne dépasse pas la capacité restante (dite résiduelle) après fixation des autres éléments, alors il est mis dans le sac. Dans le cas contraire, l'élément sélectionné se situe juste après et ainsi de suite jusqu'à épuisement de tous les objets pouvant être mis dans le sac.

Entrée : Une instance du KP.

Sortie : Une solution réalisable $\overline{x} = (\overline{x_1}, ..., \overline{x_n})$.

1. $\overline{C} := C$;
2. **pour** $i = 1$ jusqu'à n **faire**
   Si $(w_i \leq \overline{C})$ **alors**
   $\overline{x} \leftarrow 1$;
   $\overline{C} \leftarrow \overline{C} - w_j$
   Sinon
   $\overline{x} \leftarrow 0$;
   finsi
   finpour
3. Sortir avec une solution réalisable $\overline{x}$

Fig. I.1 : Heuristique gloutonne pour le KP

I.2.1.2 Calcul de bornes supérieures

Calculer des bornes supérieures ou inférieures permet d'encadrer la valeur de la solution optimale pour les problèmes que l'on tente de résoudre et de réduire ainsi l'espace de recherche des solutions. Ces bornes sont ensuite, utilisées pour le développement de méthodes de résolution exacte s'appuyant sur des procédures d'énumération implicite (ou méthodes de séparation et évaluation).

Dantzig [18] proposa une méthode efficace et élégante pour résoudre la relaxation continue du problème KP et donc determina aussi une borne supérieure pour le KP.
Le problème relâché est obtenu en éliminant les contraintes d'intégrité sur les variables x_j en les relâchant dans l'intervalle [0, 1]. Il s'écrit de la façon suivante :

$$R(KP)\begin{cases} \max Z(x) = & \sum_{j=1}^{n} v_j x_j \\ \text{s.c.} & \sum_{j=1}^{n} w_j x_j \leq C, \\ & x_j \in [0,1],\ j \in \{1,\ldots,n\}, \end{cases}$$

En supposant que les objets sont ordonnés selon l'ordre (I.1), la résolution du problème $R(KP)$ consiste alors à remplir le knapsack objet après objet et de proche en proche jusqu'à sa saturation. Ensuite, le premier objet, noté x_l $(1 < l \leq n)$, ne pouvant être mis en totalité dans le sac est repéré. Ce dernier est appelé *l'élément critique*. Il vérifie la propriété suivant :

$$\sum_{j=1}^{l-1} w_j \leq C < \sum_{j=1}^{l} w_j$$

Plus formellement, la solution $\bar{x}$ de R(KP) peut se se présenter selon la propriété de Dantzig [18] comme suit :

$$\bar{x} := \begin{cases} 1 & \text{si } j = 1,\ldots,l-1, \\ \frac{C - \sum_{j=1}^{l-1} w_j}{w_l} & \text{si } j = l, \\ 0 & \text{si } j = l+1,\ldots,n, \end{cases}$$

et la valeur de la solution optimale de $R(KP)$ est donnée par :

$$Z(\bar{x}) = \sum_{j=1}^{l-1} v_j + \bar{C}\,\frac{v_l}{w_l} \quad \text{où} \quad \bar{C} = C - \sum_{j=1}^{l-1} w_j$$

Sachant que les valeurs des v_j et x_j sont entières, la partie entière inférieure de $U(\bar{x})$, notée UB_d, définit une borne supérieure pour le KP. Cette dernière, communément appelée borne de Dantzig, s'écrit de la façon suivante :

$$UB_d = \sum_{j=1}^{l-1} v_j + \left\lfloor \bar{C} \frac{v_l}{w_l} \right\rfloor$$

Une amélioration de cette borne a été proposée par Martello et Toth [76].

I.2.1.3 Méthodes exactes

Plusieurs approches de résolution exacte pour le problème du KP ont été élaborées. La plupart de ces méthodes sont basées sur les techniques énumératives s'appuyant, généralement, sur une méthode par séparation et évaluation (Branch & Bound). Il s'agit d'énumérer d'une manière implicite les solutions et d'en choisir la meilleure parmi toutes. Le premier algorithme par séparation et évaluation a été introduit par Kolesar [49] pour résoudre le KP mais nécessitait un large temps de résolution. Un peu plus tard, d'autres versions de l'algorithme par séparation et évaluation ont vu le jour tels que l'algorithme de Greenberg and Hegerich [33] et l'algorithme de Horowitz and Sahni [41] que nous présentons dans ce qui suit.

Méthode par séparation et évaluation

La méthode par séparation et évaluation utilise deux concepts : le branchement ou la séparation du problème en sous-problèmes (représentés par des nœuds) et l'évaluation de ceux-ci à l'aide d'une relaxation (continue ou lagrangienne par exemple). L'évaluation d'un nœud consiste à borner les solutions et élaguer les nœuds inutiles, comme illustré dans la figure I.2. Notons que l'efficacité de la méthode par séparation et évaluation dépend essentiellement de la stratégie de développement qui peut être en largeur, en profondeur ou meilleur d'abord. De plus, la méthode du branchement employée et la nature de la fonction d'évaluation utilisée influencent fortement cette méthode.

L'algorithme par séparation et évaluation développé par Horowitz et Sahni considère les éléments ordonnés selon l'ordre décroissant du rapport profit par poids. Ensuite, la séparation se fait sur l'élément suivant. Depuis chaque nœud de l'arborescence, et pour un élément j pris dans l'ordre prédéfini, on développe deux branches : la première branche, $x_j = 1$ correspondant à l'élément j mis dans le sac et la deuxième branche $x_j = 0$ cor-

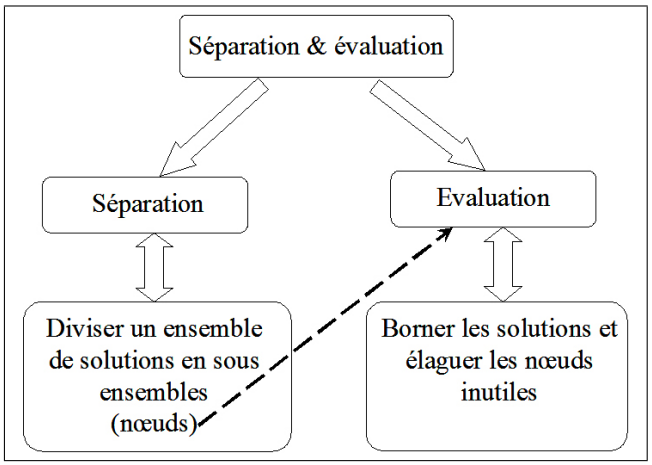

FIG. I.2 : Schéma de la méthode de Branch and Bound

respond à l'élément j qui n'est pas mis dans le sac. Le développement de l'arborescence

Entrée : Une instance d'un problème du knapsack ;

Sortie : Une solution optimale x^* ;

1. $j := 1$; MeilleureValeur := 0;
2. Calcul de la borne supérieure UBd que l'on peut atteindre depuis j ;
3. **Si** ($UBd \geq$ MeilleureValeur) **alors** ;
   Développer une branche de l'arborescence en profondeur d'abord pour obtenir une solution réalisable $\underline{x}'$;
   $j := j + 1$;
   Si ($Z(\underline{x}') \geq$ MeilleureValeur) **alors**
   $\underline{x} = \underline{x}'$;
   MeilleureValeur := $Z(\underline{x}')$;
   FinSi
   FinSi
4. $j := max\{l \ : l < j$ et $\underline{x}'_l = 1\}$
   Tant que j existe **faire**
   $\underline{x}'_j := 0$;
   Aller à 2 ;
   Fin Tant que
5. $\underline{x}' := \underline{x}$.

FIG. I.3 : Algorithme de branch-and-bound en profondeur pour le KP

se fait en profondeur, en privilégiant les branches pour lesquelles les éléments sont mis dans le sac. La fonction d'évaluation utilisée est la borne de Dantzig UBd présentée dans

la section I.2.1.2. Un élément n'est sélectionné que si son poids ne dépasse pas la capacité restante ou encore disponible du sac. Ainsi, à chaque développement d'une branche complète de l'arborescence, la solution obtenue reste réalisable. Le calcul des bornes de Dantzig se fait uniquement au niveau des nœuds développés par une branche correspondant à un élément j non sélectionné, c'est-à-dire correspondant à $x_j = 0$. Pour les nœuds développés par une branche correspondant à $x_j = 1$, la valeur de la borne de Dantzig n'est autre que celle du nœud père. Celle-ci est comparée à la valeur de la meilleure solution obtenue jusque là. Si la valeur de la borne est inférieure à la valeur de cette meilleure solution, alors il est évident qu'on ne pourra jamais atteindre une meilleure solution à partir de ce nœud. Donc, la troncature au niveau de cette branche courante est possible. La figure I.3 représente les principales étapes de cet algorithme.

Notons qu'il existe d'autres méthodes de résolution exacte pour les problèmes de type knapsack. En particulier, pour le problème du knapsack classique (KP), on trouve la programmation dynamique (Elkihel [23], Kellerer [44]) ainsi que les méthodes hybrides (voir, par exemple Boyor [10], Plateau [68]).

I.3 Problème du knapsack multidimentionnel à choix multiple (MMKP)

Dans cette partie, nous présentons la modélisation du problème du knapsack généralisé à choix multiple ou problème du knapsack multidimensionnel à choix multiple, qui est une version bien particulière du problème du KP. Il peut être considéré comme une variante qui généralise deux autres problèmes généralisant à leur tour le problème du knapsack : le problème du knapsack multidimensionnel, noté MDKP (voir Shih [75]), et le problème du knapsack à choix multiple, noté MCKP (voir Nauss [61]). Ces deux derniers ont bien été étudiés par le passé (voir Martello et Toth [56], Pisinger [67], Oliva et al [62] et Chu et Beasley [13]) pour lesquels diverses approches de résolutions exactes et heuristiques efficaces on été développées.

Le problème MMKP est caractérisé par la donnée d'un vecteur capacité ou ressources $R = (R^1, R^2, ..., R^m)$ et d'un ensemble $S = (S_1, ..., S_i, ..., S_n)$ d'objets divisés en n classes disjointes telles que pour chaque couple (p, q), $p \neq q$, $p \leq n$ et $q \leq n$, nous avons

$S_p \cap S_q = \emptyset$ et $\cup_{i=1}^{n} S_i = S$. Chaque classe $i, i = 1, ..., n$ est de cardinalité r_i (nombre d'objets de la classe i).

À chaque objet j de la classe i sont associés un profit positif v_{ij} et un vecteur poids $W_{ij} = (w_{ij}^1, ..., w_{ij}^2, ..., w_{ij}^m)$. Le but est d'attribuer au sac, exactement un et un seul objet par classe avec le maximum de profit sans violer les contraintes de capacité.

Le MMKP peut donc être formulé de la façon suivante :

$$(MMKP) \begin{cases} Z(x) = \max \displaystyle\sum_{i=1}^{n} \sum_{j=1}^{r_i} v_{ij} x_{ij} & \\ \text{s.c.} \displaystyle\sum_{i=1}^{n} \sum_{j=1}^{r_i} w_{ij}^k x_{ij} \leq R^k, & k \in \{1, \ldots, m\} \quad (1) \\ \displaystyle\sum_{j=1}^{r_i} x_{ij} = 1, & i \in \{1, \ldots, n\} \quad (2) \\ x_{ij} \in \{0, 1\}, & i \in \{1, \ldots, n\},\ j \in \{1, \ldots, r_i\}. \end{cases}$$

Notons que la variable x_{ij} vaut 1 si l'objet j de la i-ème classe est pris dans le sac et vaut 0, sinon. Les contraintes de type (1) représentent les contraintes de capacité. Les contraintes de type (2), appelées *contraintes de choix*, assurent que de chaque classe un seul objet doit être séléctionné. Sans perte de généralité et pour éviter les solutions triviales, nous formulons l'hypothèse suivante :

$$\sum_{i=1}^{n} \min_{1 \leq j \leq r_i} \left\{ w_{ij}^k \right\} \leq R^k \leq \sum_{i=1}^{n} \max_{1 \leq j \leq r_i} \left\{ w_{ij}^k \right\} \text{ pour } k = 1, \ldots, m$$

Notons aussi que lorsque le problème MMKP ne contient qu'une contrainte de capacité ($m = 1$), alors il devient un MCKP, traité dans la littérature (voir, par exemple, Dudzinski et Walukiewicz [22], Nauss [61] et Pisinger [66]). Lorsque $n = 1$ et si l'on supprime la contrainte du choix (2), alors le MMKP devient un MDKP. Divers travaux existent aussi sur ce problème (voir Pisinger [67], Chu et Beasley [13]).

Concernant le MMKP, plusieurs méthodes ont été élaborées (Moser et al [60], Khan et al [47] et [46], Hifi et al ([38], [39] et [40]), Sbihi [72] et [73]).

Dans la suite, nous présentons quelques méthodes de résolution existantes pour le problème MMKP. Nous montrerons la difficulté du problème tant dans sa résolution exacte que dans l'élaboration de méthodes heuristiques pour la construction de solutions réalisables.

I.3.1 Applications

Bien que le MMKP n'ait pas été considérablement étudié dans le passé, il intervient aujourd'hui dans diverses applications pratiques de grande importance. En effet, il peut être utilisé dans la modélisation des problèmes d'allocation des ressources dans un réseau informatique. Il peut également modéliser le problème d'adaptation dynamique des ressources de système multimédia pour assurer la qualité de service nécessaire pour le trafic multimédia (voir Khan et al [47]). Dans un tel système, plusieurs utilisateurs distants connectés par un réseau communiquent en employant voix et image aussi bien que du texte en temps-réel et la dificulté se situe au niveau de l'allocation dynamique des ressources telles que : la largeur de la bande passante (angl.bandwidth), la mémoire centrale, les cycles CPU des processeurs, etc.

Le système multimédia est dit adaptatif, lorsque les applications sont potentiellement adaptées aux ressources disponibles. Par exemple, une vidéo peut être transmise avec une cadence de 30 trame/sec selon la bande passante disponible qui peut être également réduite au besoin à la moitié en diminuant la cadence à 15 trame/sec.

L'objectif dans un système multimédia adaptatif est de construire un système multimédia où les applications reçues sont dynamiquement adaptées au statut du système, en particulier aux changements des ressources disponibles et aux préférences et exigences des utilisateurs.

Modélisation du Problème Multimédia Adaptatif (PMA)

Considérant un (PMA) comprenant plusieurs sessions concurrentes, les capacités de chaque session sont adaptées de façon dynamique aux ressources disponibles et aussi aux préférences des utilisateurs. Khan et al [47] ont proposé une modélisation mathématique de ce problème, appelé modèle-utilité. Les principaux concepts de ce modèle sont les suivants :

- Chaque utilisateur spécifie le profil qualité, qui est un ensemble de qualités acceptables pour le fonctionnement de sa session. Ces qualités sont ordonnées par ordre croissant de qualité de préférences (par exemple de vidéo noir et blanc au vidéo couleur haute définition) et exprimé par un vecteur

$$P_i = (q_{i1}, q_{i2}, \ldots, q_{ij}, \ldots, q_{il_i}),$$

où l_i est le nombre de qualités.

- Les qualités de fonctionnement d'une session sont associées aux quantités de ressources qu'elles nécessitent. Supposons que le système dispose de deux ressources :(a) la mémoire principale, (b) la bande passante du réseau. Les ressources nécessaires r_i de la session i peuvent donc s'exprimer par $r_i = (m_i, b_i)$ ou aussi par $r_i = r(q_i)$.
- Pour chaque session i, une qualité de fonctionnement q_i est associée à l'utilité de la section i, notée $u_i(q_i)$. L'utilité du système est exprimée comme suit :

$$U = \sum_{i=1}^{n} u_i(q_i)$$

- Le système doit respecter les containtes liées aux ressources. Ceci veut dire que la somme des ressources allouées à toutes les sessions ne doit pas dépasser le total des ressources disponibles. Supposons que les ressources disponibles dans le système sont représentées par le vecteur $R = (M, B)$, alors les contraintes de ressources sont représentées comme suit

$$\sum_{i=1}^{n} r(q_i) \leq R$$

Le problème de base consiste à trouver la qualité de fonctionnement q_i de chaque session i qui maximise l'utilité du système U en respectant les contraintes de ressources. Le problème (PMA) se formule en problème du knapsack multidimensionnel à choix multiple de la manière suivante :

1. Le profil qualité d'une session $P_i = (q_{i1}, q_{i2}, \ldots, q_{ij}, \ldots, q_{il_i})$ peut être considéré comme un groupe de l_i objets. Les n sessions représentent donc n groupes d'objets où chaque objet j d'un groupe i dispose d'une qualité de fonctionnement q_{ij}.
2. Lorsqu'une session i opère suivant la qualité q_{ij}, l'utilité u_{ij} de cette session est notée v_{ij} et la quantité de ressource k consommée est notée w_{ij}^k.
3. L'utilité U de (PMA) conduit à la valeur des profits V des objets choisis dans le (MMKP)
4. les contraintes de ressources de problème (PMA) sont équivalentes aux contraintes de capacités de (MMKP).
5. Trouver la qualité de fonctionnement de chaque session i peut être vu comme séléctionner exactement un élément de chaque groupe i.

I.3.2 Méthodes de résolution approchée

Plusieurs méthodes approchées ont été développées pour résoudre le MMKP. La première, proposée par Moser et al [60], se base essentiellement sur la relaxation lagrangienne. Elle est inspirée de l'heuristique développée par Magazine et Oguz [53]. Cependant, elle est développée de telle sorte qu'elle prend en compte les contraintes du choix. Elle applique un mécanisme de dégradation avantageuse des multiplicateurs de Lagrange. L'heuristique démarre en choisissant pour chaque classe l'élément du plus grand profit et en initialisant les multiplicateurs de Lagrange à zéro. En général, les contraintes de capacité seront violées. Puis, un procédé d'amélioration est répété pour la contrainte la plus violée jusqu'à obtenir une solution réalisable ou prouver qu'une telle solution n'existe pas. Le procédé d'amélioration consiste à calculer l'accroissement du multiplicateur. Cet accroissement résulte de l'échange de l'élément sélectionné avec un autre élément de la même classe. Eventuellement, l'élément correspondant au plus petit accroissement pour le multiplicateur est sélectionné pour l'échange. Ce choix minimise la différence entre la solution optimale et la solution obtenue par l'heuristique. L'heuristique a une complexité de l'ordre de $O(m(nr)^2 + mr)$ où $r = max\{r_1, \ldots, r_n\}$.

Trois autres méthodes de résolution approchée seront présentées dans les paragraphes suivants.

Une méthode d'agrégation des ressources pour le MMKP

Cette méthode heuristique à été proposée par Khan [46]. Elle utilise le concept d'agrégation des ressources de Toyoda [78] en transformant le vecteur poids d'un élément j de S_i, (c'est-à-dire, $W_{ij} = (w_{ij}^1, \ldots, w_{ij}^2, \ldots, w_{ij}^m)$), comme suit :

$$a_{ij} = \frac{\langle W_{ij}, R \rangle}{\mid R \mid}, \text{ pour} j = 1, ..., |S_i|$$

où $\langle ., . \rangle$ est le produit scalaire dans $\mathbb{R}^m$ et $\mid . \mid$ est la norme euclidienne dans $\mathbb{R}^m$. Lidée principale est de pénaliser la consommation des ressources (capacités) dépendant de l'état courant des ressources. Elle applique une forte pénalité pour les poids imposants et une faible pénalité pour les poids les moins imposants. La méthode est basée sur les points suivants :

1. Les éléments de chaque classe S_i, $(i = 1, \ldots, n)$ sont rangés dans l'ordre croissant des valeurs profits.

2. Une première solution est construite en choisissant de chaque classe S_i l'élément de plus faible profit v_{ij}. Puis, une amélioration de cette solution est envisagée, en remplaçant les éléments de plus faible profit par ceux de profit plus grand, tout en essayant de garantir la réalisabilité. Cette opération de permuter les éléments est appelée *revalorisation*. Elle est dite *revalorisation réalisable* si la solution induite est réalisable, sinon elle est non réalisable.

3. Un processus de fixation d'un élément est appliqué. Cela consiste à choisir l'élément qui maximise la valeur des ressources aggrégées disponibles. En revanche, si un tel élément ne peut être trouvé, on cherche un élément maximisant le gain par unité de profit des ressources agrégées.

Ci-après une représentation des principales étapes de l'algorithme approché, noté AppAlg, avec quelques notations :

p : vecteur position de chaque élément des classes S_i pour $i = 1, \ldots, n$ dans la solution.

$C = (c^1, \ldots, c^m)$: $m-$vecteur des ressources consommées.

Δr : ressource agrégée épargnée.

Δp : gain total par unité de ressources agrégées.

La boucle de "Étape générale" de AppAlg (voir figure I.4) tente de trouver une revalorisation réalisable. Si une telle revalorisation est possible, la solution est mise à jour et l'heuristique procède à une autre itération. Trouver une revalorisation réalisable repose sur les principes suivants :

a) trouver l'agrégation des ressources Δa de toutes les classes.

b) s'il existe au moins une revalorisation réalisable permettant de disposer des ressources agrégées, alors AppAlg sélectionne la revalorisation qui maximise l'épargne des ressources agrégées ;

c) en revanche, s'il n'existe pas de revalorisation réalisable permettant d'avoir des ressources disponibles, alors AppAlg choisit la revalorisation qui maximise le gain par unité de ressources agrégées. Notons que la complexité au pire cas de AppAlg est évaluée à $O(mn^2(r-1)^2)$, où $r = max\{r_1, \ldots, r_n\}$. (voir Khan [46]).

Une procédure constructive pour le MMKP

Cette méthode gloutonne a été proposée par Hifi et al [38]. C'est une procédure à deux

Entrée : Une instance de MMKP ;

Sortie : Une solution approchée pour le MMKP ;

Initialisation :

1. Pour(i=1,...,n) faire $p_i = 1$; /*Démarrer avec des éléments de plus petite valeur profit*/
2. $\forall k = 1, \ldots, m$, $C^k = \sum_{i=1}^{n} \sum_{j=1}^{r_i} w^k_{ip_i}$; /*Calculer le vecteur des consommations des ressources C*/

Étape générale :

Tant que(1) /*Amélioration itérative*/

- $\Delta r_{max} = 0$, $\Delta p_{max} = 0$;
- **pour**$(i = 1, \ldots, n; j = p_i + 1, \ldots, r_i)$ **faire**

  Si $(\exists k : k = 1, \ldots, m,\ C^k - w^k_{ip_i} + w^k_{ij} > R^k)$ **continue** ;

  $\Delta r = \frac{\langle (W_{ip_i} - W_{ij}).C \rangle}{|C|}$; /* Calculer l'épargne des ressources agrégées*/

  Si $\Delta r > \Delta r_{max}$ **alors**

  $\Delta r_{max} = \Delta r$, $i' = i$, $j' = j$;/* Revalorisation avec le maximum de l'épargne des ressources */

  FinSi

  Si $(\Delta r_{max} \leq 0)$ **alors**

  $\Delta p = \frac{v_{ip_i} - vij}{\Delta r}$;

  Si $(\Delta p > \Delta p_{max})$ **alors**

  $\Delta p_{max} = \Delta p$, $i' = i$, $j' = j$;/* Revalorisation avec le maximum de valeur/agrégation */

  FinSi

  FinSi

  Si $(\Delta r_{max} \leq 0$ et $\Delta p_{max} \leq 0)$

  Retourner p ;

  $C^k = C^k - w^k_{i'p_{i'}} + w^k_{i'j'}$;/* Mise à jour des ressources agrégées*/

  $p_{i'} = j'$; /* Mise à jour de la solution */

FinTantque

FIG. I.4 : Algorithme d'agrégation des ressources pour le MMKP

phases : une phase *rejet* et une phase *ajout*. Elle se résume dans les étapes principales suivantes :

1. Calculer le rapport pseudo-utilité $u_{ij} = v_{ij} / \sum_{k=1}^{m} R^k w^k_{ij},\ j \in \{1, \ldots, r_i\}$ pour tout objet j appartenant à une classe S_i.

2. Sélectionner l'élément j de chaque classe $S_i, i \in 1, ..., n$, qui réalise le plus grand rapport u_{ij}. Deux cas se présentent :
   - la solution obtenue est réalisable, alors CP s'arrête.
   - la solution n'est pas réalisable, appliquer alors la phase *rejet*. Cette dernière consiste à considèrer la contrainte la plus violée qu'on note R^{k_0} et sélectionner la classe S_{i_0} correspondant à l'élément fixé j_{i_0} ayant le plus grand poids $w^{k_0}_{i_0 j_{i_0}}$ parmi tous les éléments fixés et relativement à cette contrainte la plus violée R^{k_0}.

Cet élément est alors échangé avec un autre élément j sélectionné dans la même classe S_{i_0} et un contrôle de réalisabilité sera effectué (phase *ajout*). Si la nouvelle solution obtenue est non réalisable, elle sélectionne l'élément j'_{i_0} de plus faible poids dans la classe déjà sélectionnée, qui à son tour devient le nouvel élément fixé dans cette classe. Ce processus est réitéré jusqu'à obtention d'une solution réalisable ou réduire le taux d'irréalisabilité.

Comme nous avons décrit ci-dessus, la solution fournie par CP peut être réalisable ou non réalisable pour le MMKP. En ce qui concerne le dernier cas, on peut outrepasser la non réalisabilité de la solution, en appliquant la procédure proposée par Hifi et al dans [38], qui est une procédure constructive complémentaire qu'on notera ici CCP. Elle utilise une stratégie de permutation locale (appelée stratégie 2-opt) entre deux objets j et j' appartenant à la même classe S_i. Notons, que plusieurs permutations peuvent être utilisées pour, d'une part, obtenir une solution réalisable et d'autre part l'améliorer de plus.
Notons aussi que les procédures CP et CCP ont une complexité au pire égale à $O(max\{\theta m, n\})$ où $\theta = max\{r_1, \ldots, r_n\}$.

Une recherche locale réactive pour le MMKP

Dans cette partie, nous décrivons la méthode de recherche locale réactive, notée MRLS, proposée par Hifi et al [39] pour résoudre le MMKP. Cette technique est une hybridation entre la recherche tabou et d'autres stratégies de recherche locale. En général, une recherche locale consiste à partir d'une solution réalisable et tenter de l'améliorer tout en répétant ce processus un certain nombre de fois. Elle est aussi souvent complétée par un *processus réactif*, s'appuyant sur l'historique de la recherche pour augmenter son efficacité. Dans l'algorithme de Hifi et al [39], la répétition de certaines configurations (solutions) est mise *tabou* au cours de la recherche. L'algorithme est basé principalement sur les points suivants :

(i) Démarrer avec une solution réalisable.

(ii) Utiliser une *stratégie de dégradation* (la procédure "dégrader()" décrite plus loin) sur une solution courante pour trouver une nouvelle solution. Le but est de changer le trajectoire de la recherche lorsqu' il n' y a pas eu d'amélioration et d'introduire

ainsi une sorte de diversification de la recherche.

(iii) Introduire une liste dite *mémoire* pour interdire toute répétition dans les configurations visitées auparavant.

Les étapes principales de l'algorithme MRLS sont décrites dans la figure I.5.
L'algorithme commence par construire une solution initiale, notée $x^* := x$ de valeur objectif $Z(x^*)$ en appliquant la procédure CP. La liste mémoire *Liste* est initialisée à l'ensemble vide. Ensuite, l'algorithme fait appel à la procédure constructive complémentaire CCP dans le but d'améliorer la solution courante si cette dernière n'appartient pas à *Liste*. La solution trouvée x' est comparée ensuite à la meilleure solution enregistrée x^*.

Entrée : Une solution réalisable x de valeur objectif Z(x).
Sortie : Une solution améliorée x^* de valeur objectif $Z(x^*)$.

1. $x^* := x := CP_{sol}$, $Liste = \emptyset$;
2. **Tant que** (condition-d'arrêt-non-vérifiée) **faire**
   $p := 0$ /* p : nombre de dégradations d'une solution */
   Répéter
   $x' = CCP_{sol}(x)$ et $x' \notin Liste$;
   Si $(V(x^*) > V(x'))$ **alors**
   $x^* := x'$; $V(x^*) := V(x')$; $p := 0$;
   Sinon
   $p := p + 1$;
   Si $(p < Const)$ **alors**
   Insérer x^* dans $Liste$;
   $x :=$ Dégrader(x^*) et $x \notin Liste$;
   FinSi
   jusqu'à $(p = Const)$
   finTant que
3. Sortir avec une solution réalisable $\overline{x}$

FIG. I.5 : Méthode de recherche locale réactive : MRLS

En cas d'amélioration, une mise-à-jour de la solution est prévue. Dans le cas contraire la solution x' est introduite dans la liste tabou ($Liste$) pour éviter un éventuel cyclage. De plus, l'algorithme prévoit d'utiliser un certain nombre de dégradations de la solution afin d'aller vers une nouvelle solution et diversifier la recherche.

Notons par $x^* = (x_1^*, \ldots, x_i^*, \ldots, x_n^*)$ la solution courante à dégrader, la procédure "dégrader()" peut être décrite comme suit :

1. Séléctionner une classe S_i arbitrairement.

2. Échanger deux éléments de la classe S_i et obtenir une nouvelle solution réalisable.

3. Répéter les étapes 1- 2 un certain nombre de fois.

Enfin, l'algorithme s'arrête après avoir atteint un nombre maximum d'itérations. La complexité de ce dernier est de $O(\theta^2(m+n))$ dans le pire des cas.

I.3.3 Méthodes de résolution exacte

Dans la littérature, il existe très peu d'algorithmes exacts traitant le MMKP. Ces algorithmes sont basés sur les méthodes par séparation et évaluation et diffèrent par la fonction d'évaluation utilisée ainsi que par la méthode de séparation. Le premier algorithme de ce type a été proposé par Khan et al [47]. Il se base sur la borne supérieure produite par le simplexe et emploie un parcours par "le meilleur d'abord" pour l'exploration de l'arbre de recherche. L'algorithme produit une solution optimale pour des instances pouvant contenir jusqu' à 300 éléments répartis sur 10 classes (voir Khan et al [47]) pour plus de détails. Le deuxième algorithme a été proposé par Sbihi [72] et [73]. Il utilise une nouvelle borne supérieure, dérivée de la borne de Dantzig [18] pour le knapsack unitaire, comme évaluation. L'une des particularités de cet algorithme concerne la séparation qui se fera en créant à partir d'un noeud donné, un noeud fils et un noeud frère tout en choisissant une exploration par le meilleur d'abord. Dans ce qui suit, nous présentons une description plus détaillée de cet algorithme et de la borne supérieure qu'il utilise.

Les bornes supérieures pour le MMKP

Sbihi [72] et [73], et Hifi et al [40] ont proposé des bornes supérieures pour le MMKP en se basant principalement sur le problème du knapsack à choix multiple (Multiple-Choice Knapsack Problem : MCKP-voir Nauss [61]). Ils l'ont considéré comme un problème auxi-

liaire du MMKP, qu'on notera ici par $MMKP_{aux}$. Ce dernier s'écrit de la façon suivante :

$$(MMKP_{aux}) \begin{cases} Z(x) = \max \sum_{i=1}^{n} \sum_{j=1}^{r_i} v_{ij} x_{ij} & \\ \text{s.c.} \sum_{i=1}^{n} \sum_{j=1}^{r_i} w_{ij} x_{ij} \leq R, & (1) \\ \sum_{j=1}^{r_i} x_{ij} = 1, \quad i \in \{1, \ldots, n\} & (2) \\ x_{ij} \in \{0,1\}, \quad i \in \{1, \ldots, n\},\ j \in \{1, \ldots, r_i\}. & \end{cases}$$

où $R = \sum_{k=1}^{m} R^k$ et $w_{ij} = \sum_{k=1}^{m} w_{ij}^k$. On pose également $NR = \sum_{i=1}^{n} r_i - n$. Dans Hifi et al [40], les auteurs ont proposé une borne supérieure pour le problème $MMKP_{aux}$ et ont démontré ensuite qu'elle est aussi une borne supérieure pour le MMKP. Soit UB cette borne calculée comme suit :

1. Repérer dans chaque classe $i,\ i = 1, ..., n$, l'élément j_{max} ayant le plus grand rapport profit par poids $(\frac{v_{ij}}{w_{ij}}),\ j = 1, ..., r_i$.
2. Soient $W_{max} = \sum_{i=1}^{n} w_{ij_{max}}$ (réspectivement $V_{max} = \sum_{i=1}^{n} v_{ij_{max}}$) la valeur des poids cumulés (respectivement les profits cumulés) des éléments j_{max} de chaque classe i. Deux cas se présentent :

   a) $W_{max} > R$. Dans ce cas la valeur de UB est donnée par :

   $$UB = \sum_{i=1}^{n} v_{ij_{max}} \times \left(\frac{R}{\sum_{i=1}^{n} w_{ij_{max}}} \right) = V_{max} \times \left(\frac{C}{W_{max}} \right).$$

   b) $W_{max} < R$. Dans ce cas, omettre pour chaque classe i, les éléments j_{max} et trier les NR éléments restants par ordre décroissant du profit par poids sans distinction de classes. Ces éléments (indexés par $j,\ j = 1, ..., NR$) forment un problème du knapsack ordinaire (KP) de capacité $C - V_{max}$, dont le profit et le poids d'un élément j, sont notés par v_j et w_j respectivement.

   Dans ce cas, la valeur de UB est donnée par :

   $$UB = V_{max} + UB_{KP}$$

   où UB_{KP} est une borne supérieure pour KP, calculée par la méthode de Dant-

zig [18]. Elle s'écrit de la façon suivante :

$$UB_{KP} = \sum_{j=1}^{l-1} v_j + \left(\frac{R - W_{max} - \sum_{j=1}^{l-1} w_j}{w_l} \right) \times v_l$$

où l est l'élément critique vérifiant :

$$l = min\{j : \sum_{i=1}^{n} w_j > R - W_{max}\}.$$

Dans Sbihi [73], la borne UB est employée comme une fonction évaluation des nœuds dans l'algorithme de séparation et évaluation que nous dévrivons dans le paragraphe suivant.

Une méthode par séparation et évaluation progressive pour le MMKP

Cette méthode a été proposée par Sbihi [72], [73]. Elle s'appuie sur (i) la génération d'une solution de départ, (ii) le calcul de la borne supérieure UB aux différents niveaux de l'arborescence, et (iii) l'application de la stratégie par le meilleur d'abord.

Entrée : Une instance du MMKP ;
Sortie : Solution optimale de MMKP ;

1. *Initialisation*
   - LB ; /* une borne inférieure pour le MMKP */
   - Trier les éléments de chaque classe dans l'ordre décroissant des profits ;
   - L = $\{n_{11}\}$; /* noeud racine de l'arborescence */
2. *Étape générale*
   Tant que $(L \neq \emptyset)$ **faire**
   (a) noeud = meilleur noeud de L ;
   (b) $L \longleftarrow L \backslash$ {noeud} ;
   (c) **Si** (noeud correspond à une solution réalisable) **alors** développer un nouveau noeud fils ;
   (d) **Si**(noeud fils appartient à la dernière classe et la solution est réalisable) **alors** aller à 3 ;
   (e) **Si**(UB(noeud fils) $> LB$) **alors** $L \longleftarrow L \cup$ {noeud fils} ;
   (f) **Si**(noeud n'est pas le dernier élément d'une classe) **alors** développer un noeud frère et le mettre dans L ;
   Fin Tant que
3. Sortir avec la solution.

Fig. I.6 : Algorithme de Branch & Bound pour le MMKP

L'algorithme commence par calculer une solution de départ, notée LB, à partir de l'algorithme MRLS proposé par Hifi et al [39] (voir aussi la figure I.5). Cette valeur constitue une borne inférieure pour le problème, utilisée pour l'évaluation des nœuds générés.
Chaque nœud développé dans l'arborescence correspond à un élément mis dans le sac. Nous notons par n_{ij} le nœud correspondant à l'élément j de la classe i suivant l'ordre décroissant des profits v_{ij}.

Au cours du développement de l'arborescence et pour un nœud n_{ij}, deux noeuds sont développés :

- Le nœud frère n_{ij+1} qui correspond, s'il existe, à l'élément suivant $j+1$ de la même classe i ;
- Le nœud fils n_{i+1j} qui correspond, si la classe suivante $i+1$ existe, au premier élément de cette même classe ;

Notons que chaque nœud de l'arbre correspond à une solution partielle de valeur $\hat{Z}$, composée des éléments déjà mis dans le sac. Notons aussi qu'en chaque noeud de l'arborecence la borne supérieure UB décrite précédemment est calculée et le nœud est ensuite évalué de telle sorte qu'il sera troncaturé dans les deux cas suivants :

1. Si $UB + \hat{Z} < LB$, puisque toute solution réalisable développée à partir de ce nœud aura une valeur strictement inférieure à LB.
2. Si le noeud est développé à partir d'un noeud père non réalisable (correspond à une solution non réalisable).

L'algorithme, décrit dans la figure I.6, se termine dès qu'une solution réalisable est obtenue. En effet, c'est aussi une solution optimale pour le problème.

I.4 Méthodes de la programmation linéaire adaptées à la résolution du MMKP

La programmation linéaire est un outil très puissant de la recherche opérationnelle. Il permet de résoudre les problèmes d'optimisation dont la fonction objectif et les contraintes sont linéaires. Lorsque les domaines des variables sont discrets, on parle alors des problèmes d'optimisation combinatoire dont le MMKP fait partie. Ces problèmes sont très difficiles à résoudre vu leur nature combinatoire qui nécessite un calcul exhaustif notamment

pour les problèmes de grande taille (nombre de variables et/ou contraintes trop élevé). Toutefois, il existe des méthodes robustes specialisées dans la résolution de ces problèmes telles que la méthode branch-and-price employant la technique de génération de colonnes et la méthode branch-and-cut employant la méthode de génération de contraintes. Dans le chapitre qui suit, nous allons présenter de façon générale ces méthodes sur lesquelles nous nous appuyons principalement pour résoudre le MMKP.

I.5 Conclusion

Dans ce chapire, nous avons présenté un tour d'horizon de quelques problèmes de type knapsack. Nous avons entamé notre présentation par le problème du knapsack classique (unitaire) puis nous avons discuté une de ses variantes qui est le problème du knapsack multidimensionnel à choix multiple. Nous avons cité certains domaines d'application de ces problèmes et nous avons présenté quelques méthodes de résolution approchée et exacte qui leurs sont associées. Puis nous avons parlé des méthodes qu'on a adaptées dans cette thèse pour résoudre le problème de knapsack multidimensionnel à choix multiple. Ces méthodes, consacrées principalement à résoudre les problèmes d'optimisation combinatoire de grande taille, sont basiques dans la programmation linéaire et la programmation linéaire en nombres entiers. Elles feront l'objet du chapitre suivant.

Chapitre II

Techniques et méthodologie de la programmation linéaire

Dans ce chapitre nous présentons un recueil des méthodes basiques de la programmation linéaire. Nous commençons par des définitions et un bref rappel sur l'algorithme de Simplexe. Puis nous décrivons la méthode de génération de colonnes et nous détaillons la méthode du "Branch-and-Price". Nous présentons ensuite un tour d'horizon de méthode des plans de coupes et les concepts de base de la théorie polyèdrale et nous terminerons ce chapitre par la présentation d'un algorithme simplifié de "Branch-and-Cut".

II.1 Introduction

La programmation linéaire représente une partie importante de la programmation mathématique. Elle consiste à modéliser des problèmes issus d'applications réelles en un langage formel et présente ensuite des méthodes pour les résoudre.
Durant la seconde guerre mondiale, George Dantzig formula en tant que problèmes linéaires (PL), de nombreux problèmes militaires, relatifs à l'organisation et à la distribution des munitions et des vivres, ainsi qu'aux déploiements des troupes militaires. Puis, en 1947, il mit au point un algorithme pour résoudre les problèmes linéaires, connu sous le nom de l'algorithme de Simplexe. Ce dernier, est considéré comme une méthode de points-frontières puisqu'il projette de faire des déplacements le long des arêtes du polyèdre défini par les contraintes du PL jusqu'à atteindre l'optimum. La découverte et l'implémentation

de l'algorithme de Simplexe, à cette époque, contribua grandement au renforcement de la valeur de la programmation mathématique. Cependant, s'il se révèle très efficace en moyenne, Klee et Minty [48] montrèrent en 1972 que cet algorithme n'est pas polynomial. Ceci a suscité un regain d'intérêt pour la recherche d'algorithmes polynomiaux de résolution des programmes linéaires.

La méthode ellipsoïde, developpée en 1979 par khachian [45], fut le premier algorithme de ce type montrant alors que les problèmes de programmation linéaires sont polynomiaux. Son idée principale consiste à utiliser une suite d'ellipsoïdes de volume décroissant mais contraint de contenir la solution optimale du problème à résoudre à chaque itération. Bien que plus rapide que l'algorithme de Simplexe sur les problèmes de Klee et Minty, l'algorithme ellipsoïde reste bien plus lent sur les problèmes réels.

En 1984, un autre algorithme polynomial basé sur les principes de géométrie projective et de programmation non linéaire à été développé par Karmakar [43]. Il s'agit d'une méthode de points intérieurs dont le principe est de chercher des points, à l'intérieur du polyèdre des contraintes, permettant de se diriger rapidement vers le sommet optimal.

Depuis la découverte de cet algorithme, la recherche dans le domaine de la programmation mathématique a connu un nouvel élan et plusieurs algorithmes tels que la méthode de barrière dérivée de ce dernier virent le jour. Actuellement, ces méthodes commencent à concurrencer l'algorithme de Simplexe sur certains problèmes de grande taille car elles ont tendance à nécessiter moins d'itérations que ce dernier bien que chaque itération soit plus longue.

Néanmoins, l'algorithme de Simplexe demeure, non seulement un outil très performant pour la résolution des PL mais, en outre, il se distingue exclusivement par la richesse de l'interprétation géométrique et économique qu'il fournit ainsi qu'à sa capacité à fournir des solutions de base, très importantes dans les approches de décompositions ou encore dans des procédures de ré-optimisation itératives (Lebbar [52]).

Dans ce chapitre nous présentons quelques définitions élémentaires de la programmation linéaire ainsi qu'un bref rappel sur l'algorithme de Simplexe. Puis, nous décrivons la méthode de génération de colonnes qui dérive de ce dernier et qui est destinée à résoudre des programmes linéaires contenant un nombre important de variables. Nous discutons également la méthode de génération des contraintes qui est une version duale de la mé-

thode de génération de colonnes utilisée pour traiter les programmes linéaires contenant un grand nombre de contraintes. Nous présentons par la suite l'approche polyédrale pour les problèmes en nombres entiers et nous terminons notre présentation par certaines techniques exactes de résolution des problèmes linéaires en nombres entiers : la méthode de "Branch and Price" et la méthode de "Branch and Cut".

II.2 Définitions

Programme linéaire

Un programme linéaire est un problème d'optimisation consistant à maximiser ou minimiser une fonction linéaire dite fonction objectif sous contraintes linéaires exprimées sous forme d'équations ou d'inéquations. Un tel programme peut être représenté sous forme matricielle comme suit :

$$(P_0)\begin{cases} \max\ z = & c.x \\ \text{s.c.} & Ax \leq b \\ & x \geq 0 \end{cases}$$

avec les notations suivantes :

- n : nombre de variables,
- m : nombre de contraintes,
- $A = (a_{ij})_{i=1,...,m\ \ j=1,...,n}$: matrice réelle $(m \times n)$ des contraintes,
- $c = (c_1, ..., c_n)$: $n-$vecteur ligne des coûts (profits),
- $b = (b_1, ..., b_m)^T$: $m-$vecteur colonne des seconds membres,
- $x = (x_1, ..., x_n)^T$: $n-$ vecteur colonne de variables.

Forme standard d'un programme linéaire

Tout programme linéaire peut être transformé pour ne comporter que des équations, en ajoutant une variable s_i appelée variable d'écart pour chaque contrainte, le programme linéaire s'écrit alors de la manière suivante :

$$(P_1)\begin{cases} \max\ z = & c.x \\ \text{s.c.} & \left(A + I \right) \begin{pmatrix} x \\ s \end{pmatrix} = b, \quad \begin{pmatrix} x \\ s \end{pmatrix} \in IR_+^{n+m} \\ & x \geq 0 \end{cases}$$

où I est la matrice identité $m \times m$, et s est le vecteur des variables d'écarts. On dit alors que le programme linéaire est mis sous *forme standard* sachant que les deux programmes linéaires (P_0) et (P_1) sont équivalents. Notons aussi que la matrice A définissant les contraintes du programme linéaire est désormais de dimension $m \times (m+n)$ et que la dimension du vecteur coût c est $m+n$ tel que les m derniers éléments de c sont nuls.

Dans la suite, nous nous intéressons uniquement aux programmes linéaires exprimés sous forme standard.

II.3 Solutions d'un programme linéaire

Une solution d'un programme linéaire est une affectation de valeurs aux variables du problème, elle est dite réalisable lorsqu'elle satisfait toutes les contraintes du problème. Une solution notée par x^* est optimale lorsqu'elle est réalisable et la fonction objectif z atteint sa valeur maximale z^*. Cette solution n'est pas nécessairement unique.
Considérons le système :

$$(P_2)\begin{cases} \max\ z = & c.x \\ \text{s.c.} & Ax = b \\ & x \geq 0 \end{cases}$$

On appelle base toute sous-matrice carrée $(m \times m)$ inversible de A. Soit B une telle sous-matrice. Alors, une permutation des colonnes de A, permet de la mettre sous la forme $A = [B,\ N]$ où N est la sous-matrice formée par les colonnes de A qui ne font pas partie de la base.
Soit x_B le vecteur des variables de base (associées à la base B) et x_N le vecteur des variables hors base, on peut donc écrire :

$$Ax = b \Longleftrightarrow B.x_B + N.x_N = b$$

On peut alors déterminer l'ensemble des solutions du système $A.x = b$ en fixant arbitrairement les valeurs de x_N et en calculant les valeurs résultantes de x_B.
Lorsque les variables hors-base x_N sont fixées à 0, une solution particulière associée à la base B est obtenue. Cette solution est appelée solution de base ; elle est exprimée comme suit :

$$x_B = B^{-1}.b \text{ et } x_N = 0$$

Une base est dite dégénérée si le vecteur $x_B = B^{-1}.b$ a une de ses composantes nulle. Si $B^{-1}.b \geq 0$ alors la base B est dite réalisable et la solution de base associée est aussi réalisable.

Le vecteur $y = c_B^T.B^{-1}$ est la solution duale de base associée à B.

Problème dual

À chaque programme linéaire (P_1) nommé programme linéaire primal, est associé un programme linéaire dual (D_1), dans lequel on doit trouver un vecteur $y \in R^m$, tel que :

$$(D_1) \begin{cases} \min\ z = & y.b^T \\ \text{s.c.} & A^T y = c \\ & y \geq 0 \end{cases}$$

où le vecteur $y = (y_1, ..., y_m)^T$ est un $m-$ vecteur ligne appelé vecteur des variables duales tel que pour chaque variable y_i $(i = 1, \ldots, m)$ est associée une contrainte i du problème (P_1). Notons qu'il existe plusieurs liens entre le problème primal (P_1) et son problème dual (D_1) dont nous citons quelques uns dans les théorèmes suivants :

Théorème II.1. *Etant donnés deux programmes linéaires duaux (P_1) et (D_1).*

- *Si (P_1) n'admet pas de solution réalisable alors soit (D_1) n'admet pas de solution réalisable, soit (D_1) est non borné.*
- *Si B est une base de (P_1), alors le vecteur $y = c_B^T.B^{-1}$ est la solution duale de base associée à B.*

Théorème II.2. (théorème de dualité)

- *Si (P_1) et (D_1) ont des solutions, alors chacun d'eux a une solution optimale et : $z^* = Max(P_1) = Min(D_1) = w^*$*
- *Si (P_1) est non borné alors (D_1) n'admet pas de solution réalisable.*

Le *théorème de dualité* est l'un des résultats fondamentaux de la programmation linéaire les plus importants, vu son intérêt théorique et pratique. Pour plus de détails sur la programmation linéaire, le lecteur est encouragé à consulter Chvàtal [15].

Algorithme de Simplexe

La méthode de Simplexe est une méthode exacte itérative qui consiste à déterminer un

premier programme de base puis à construire une suite de programmes de base réalisables améliorant constamment la fonction économique et donc conduisant à l'optimum. Géométriquement, le simplexe consiste à se déplacer d'un point extrême à l'autre, le long des arêtes du polyèdre défini par les contraintes, jusqu'à trouver le point associé à la solution optimale.

Considérons le programme linéaire (P2) de la section II.3 et soit B une base réalisable (non dégénérée) issue de la matrice des contraintes. On rappelle qu'on peut écrire la matrice des contraintes $A.x = b$ de la façon équivalente $B.x_B + N.x_N = b$.
Sachant aussi que la fonction objectif peut se décomposer ainsi :

$$z = c.x = c_N.x_N + c_B.x_B,$$

nous disposons alors de l'écriture suivante du problème (p2), qu'on appelera *forme canonique* :

$$\begin{aligned} x_B &= B^{-1}.b - B^{-1}.N.x_N \\ z &= c_B.B^{-1}.b + (c_N - c_B.B^{-1}.N).x_N \\ x_B &\geq 0,\ x_N \geq 0. \end{aligned}$$

z s'écrit aussi de la façon suivante : $z = c_B.B^{-1}.b + (c_N - y.N).x_N$ où $y = c_B.B^{-1}$ est le m-ligne vecteur des variables duales.
Posons $\hat{c} = c_N - y.N$, $\hat{c}$ est appelé vecteur des coûts réduits (marginaux).

À chaque itération de l'algorithme de Simplexe, une variable de x_B dite *sortante* doit être échangée avec une variable de x_N dite *entrante*, de façon à obtenir une nouvelle base réalisable et augmenter la valeur de l'objectif. Dans le cas où on cherche un maximum, l'existence d'une variable hors base x_j ayant un coût réduit positif $\hat{c_j}$ montre que la base B courante n'est pas optimale et qu'une augmentation de l'objectif est encore possible. Pour ce faire, une opération appelée *pivotage* devient nécéssaire. Elle consiste à augmenter la variable x_j dans la base jusqu'a l'annulation d'une autre variable de la base. Ensuite, la base doit être actualisée en remplaçant la variable annulée (sortante) par la variable x_j (entrante) dans l'ancienne base B, ce qui donne lieu à une nouvelle base B' correspondant géométriquement à un autre point extrême dans le polyèdre des contraintes. Ce processus est réitéré jusqu'à ce que tous les coûts réduits soient négatifs ou nuls. Pour plus de détails sur la méthode de Simplexe, notamment sur la prise en compte de ces cas spéciaux et sur l'étude de sa complexité, on peut se référer à des ouvrages traitant de la programmation

linéaire tels que (Minoux [58], Teghem [77], Goldfarb & Todd [31]).

II.4 Programmes linéaires de grande taille

De nombreux problèmes d'optimisation combinatoire peuvent être modélisés comme des PLNE, mais leur taille, ainsi que leur structure empêchent fréquemment une résolution par les méthodes usuelles de la programmation linéaire. En effet, la relaxation linéaire de ces modèles fournit souvent une borne de mauvaise qualité qui, ajoutée à la combinatoire du problème, rend la recherche arborescente par séparation et évaluation d'une solution optimale entière, très difficile voir même impossible en un temps raisonnable. Une solution à ce problème consiste à utiliser des méthodes de décomposition (décomposition de Benders [7], décomposition de Dantzig-Wolfe [17]) donnant lieu à de nouveaux modèles plus efficaces et fournissant de meilleures bornes. En contrepartie, ces modèles nécessitent l'utilisation d'algorithmes complexes : génération de coupes pour la décomposition de Benders (nombre exponentiel de contraintes), génération de colonnes pour la décomposition de Dantzig-Wolfe (nombre exponentiel de variables). Notons que le choix de la méthode de décomposition employée dépend strictement de la stucturation du problème linéaire c'est-à-dire la nature de ses composantes (contraintes et variables) et les liens existants entre chacune d'elles.

En général, les programmes linéaires de grande taille sont caractérisés par leurs matrices de contraintes qui sont "très creuses" et les éléments nuls sont distribués de telle façon qu'ils forment de grandes sous-matrices (voir Minoux [58] pour plus de détails).

La décomposition de Dantzig consiste à partitionner les contraintes d'un PL de façon à obtenir des sous-problèmes indépendants, liés entre eux uniquement par un ensemble de contraintes couplantes. Cependant, le modèle obtenu par cette formulation alternative est de trop grande taille pour être totalement explicité et donc résolu par un algorithme tel que l'algorithme de Simplexe. Cependant, cette formulation offre la possibilité de se ramener à une suite de sous-problèmes plus faciles à résoudre que le problème global et dont les liens de dépendance sont traités à un niveau supérieur de coordination. Cette technique de résolution des programmes linéaires de grande taille, appelée génération de colonnes, fait l'objet des paragraphes suivants.

II.4.1 Principes de résolution avec la génération de colonnes

La méthode de génération de colonnes, ou programmation linéaire généralisée, permet de résoudre des problèmes contenant un nombre important de variables interdisant l'application de l'algorithme de Simplexe au problème dans sa globalité. L'idée centrale de cette technique vient du fait que lors de la résolution d'un programme linéaire par la méthode de Simplexe, plusieurs variables sont hors base et, très souvent, la plupart d'entre elles sont nulles à l'optimum, et peuvent donc être négligées.
Reposant sur cette particularité, la génération de colonnes consiste à ne manipuler, à la fois, qu'un sous ensemble de variables (colonnes) de petite taille et à identifier les variables entrant en base au cours de la résolution sans les énumérer explicitement. En effet, le mécanisme de génération de colonnes, s'effectue au sein d'un algorithme, appelé *algorithme générateur* ou aussi *"algorithme de pricing"* qui résout généralement un ensemble de sous problèmes indépendants afin de filtrer les meilleures colonnes hors formulation actuelle du programme linéaire, c'est-à-dire celles qui sont susceptibles d'améliorer la solution courante. Ces colonnes correspondent aux variables dont les coûts réduits sont positifs (problème de maximisation). Autrement dit, la génération de colonnes est basée sur la décomposition du problème original (LP) en un problème maître et un sous-problème. Le problème maître contient seulement un sous-ensemble de variables et le sous-problème est résolu pour déterminer si des colonnes peuvent être rajoutées au problème maître ou alors affirmer le contraire.
La technique de génération de colonnes a été introduite par Gilmore et Gomory entre 1961 et 1965 ([28], [29] et [30]) pour résoudre le problème de découpe industrielle. Dans leurs travaux, l'algorithme générateur consistait à résoudre un problème du knapsack qui représentait des patrons de découpe. On trouve une description détaillée de ce problème dans un grand nombre d'ouvrages traitant la génération de colonnes (par exemple Lasdon [51], Goldfard & Todd [31], Minoux [59] et Desaulniers [20]). Ensuite, la méthode a prouvé son efficacité à résoudre de nombreux problèmes d'optimisation combinatoire tels que le problème de coloration de graphes (voir Mehrotra et Trick [57], les problèmes d'affectation et d'ordonnancement de tâches (Savelsbergh [71], Van Den Akker et al [79]), les problèmes de tournées de véhicules avec contraintes supplémentaires -fenêtres de temps et contraintes de ressources- (Desrochers et al [21] et Ribeiro & Soumis [69]) et les problèmes

d'optimisation combinatoire issus d'applications spatiales (voir Mancel [54]).

Considérons le problème linéaire (PL), écrit sous sa forme standard comme suit :

$$(PL)\begin{cases} \max\ z = & c.x \\ \text{s.c.} & Ax = b \\ & x \geq 0 \end{cases}$$

On suppose que (PL) comporte un nombre raisonnable m de contraintes et un très grand nombre de variables n tels que $n \gg m$ et la matrice A ne peut être représentée explicitement mais que l'on dispose d'un outil pour caractériser ses colonnes. On notera par N, l'ensemble de variables de (PL).

Considérons ensuite un sous-ensemble Ω des variables de (PL) et notons par A' la sous-matrice de A correspondant à Ω et par $\widehat{c_j}$ le coût réduit d'une variable hors base x_j, défini par :

$$\widehat{c_j} = c_j - \alpha A_j$$

où α est le vecteur des multiplicateurs de Simplexe (variables duales) associés à la base optimale du problème restreint courant.

La méthode de génération de colonnes est décrite dans l'algorithme de la figure II.1.

Notons que l'efficacité de la méthode de génération de colonnes est très dépendante de l'algorithme générateur. En effet, ce dernier revient souvent à résoudre des sous-problèmes, qui sont NP-difficiles. Une résolution approchée peut être envisagée dans ces cas.

Notons enfin que la génération de colonnes peut être également vue comme une version duale de la relaxation Lagrangienne. On peut trouver une description détaillée ainsi que des éléments de comparaison avec la génération de colonnes dans différents ouvrages, en particulier dans (Minoux [59]).

II.4.2 La méthode de génération de contraintes

Les programmes linéaires qui contiennent un très grand nombre de contraintes ne peuvent pas être représentés explicitement en mémoire ou tenir dans un solveur de programmes linéaires. Pour les résoudre, une méthode de résolution semblable à la génération de colonnes existe. Il s'agit de la méthode de *génération de contraintes* ou *méthode des plans de coupe* qui consiste à choisir un sous-ensemble de contraintes, qu'on notera ω, du

Entrée : Une instance du problème (PL) ;

Sortie : Une solution optimale de (PL) ;

1. **Initiation :** $B_0 :=$ base réalisable initiale ; Iteration k = 0 ; pricing = vrai.
2. **Tant que** (pricing=vrai) **faire**
   (a) Calculer la solution de base $\overline{x} = {B_k}^{-1}.b$;
   (b) Calculer les variables duales $\alpha = c_{B_k}.{B_k}^{-1}$;
   (c) Résolution du *problème de "pricing"* :
   Pour ($j \in N \setminus \Omega$)
   i. **Si** $\widehat{c_j} = c_j - \alpha A_j \leq 0$ **alors**
   pricing = faux ;
   ii. **Sinon**
   Ajouter la variable j avec $\widehat{c_j} > 0$ à Ω ;
   Ajouter la colonne A_j à A' ;
   pricing = vrai ;
   (d) $k = k + 1$.
   Fin Tant que
3. Sortir avec une solution optimale de (PL).

Fig. II.1 : Algorithme de génération de colonnes : GC

problème (PL) qu'on veut traiter et résoudre la relaxation du problème restreint (PLR) qui en résulte. Puis, essayer d'ajouter itérativement des contraintes du problème initial, appelées des *plans de coupe*, violées par la solution courante jusqu'à ce qu'il n'y en ait plus. Dans ce cas, l'algorithme s'arrête et la solution courante constitue une solution optimale du problème original (LP). La méthode de génération de contraintes est illustrée dans la figure II.2.

Notons que l'ensemble des solutions réalisables du problème (PLR) relaxé contient celui de (PL). Ainsi, pour un problème de maximisation la valeur de la fonction économique d'une solution optimale de (PLR) est supérieure ou égale à celle d'une solution optimale du problème original (PL). Notons aussi que l'algorithme de génération de contraintes n'impose pas d'avoir une liste exhaustive explicite des plans de coupes, mais seulement une méthode pour générer efficacement des contraintes valides violées. L'identification de ces contraintes est appelée *le problème de séparation*.

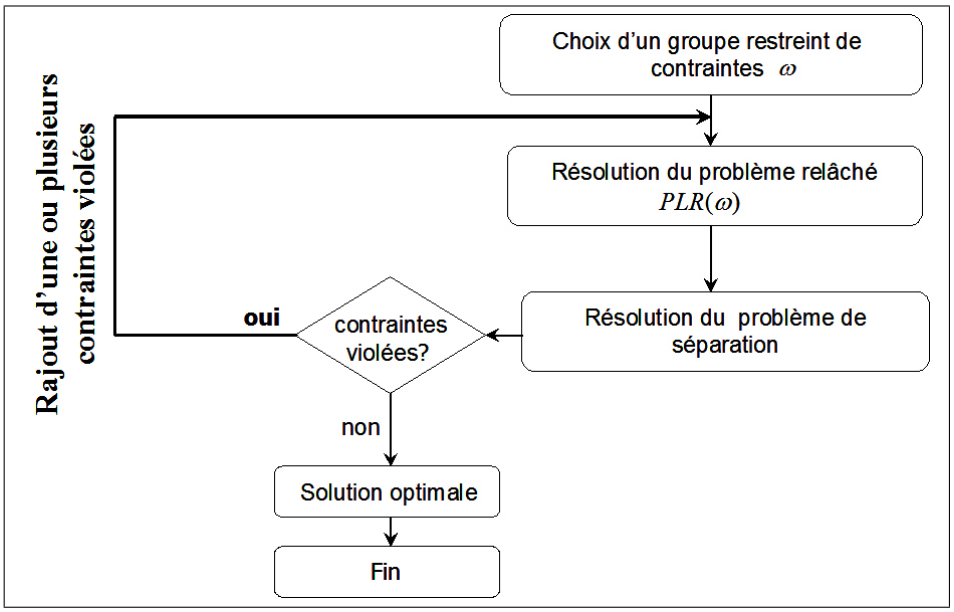

Fig. II.2 : Schèma général de la méthode de génération de contraintes

Problème de séparation :
Etant donné un ensemble de contraintes d'un programme linéaire et un vecteur $x \in \mathbb{R}^n$, le problème de séparation vise à prouver que x satisfait toutes les contraintes ou bien trouver une inéquation violée par x.

II.4.3 Programmation linéaire en nombres entiers

Plusieurs problèmes d'optimisation combinatoire se formulent sous forme de programmes linéaires en nombres entiers (PLNE) dont une grande partie appartient à la classe des problèmes NP-difficiles. Pour certains cas particuliers de ces problèmes, dont la matrice des contraintes est totalement unimodulaire, la solution optimale de la relaxation linéaire du problème est entière. Par conséquent, les algorithmes classiques de programmation linéaire peuvent être utilisés pour les résoudre. En dehors de ces cas particuliers, la solution optimale de la relaxation linéaire d'un PLNE fournit une borne supérieure (problème de maximisation) de la valeur optimale. La méthode de résolution des PLNE qui semble la plus efficace est la méthode par séparation et évaluation présentée dans le chapitre I. Cependant, cette méthode peut s'avérer infructueuse lors de la résolution de problèmes de taille importante. En conséquence, on fait souvent appel à des méthodes conjuguant les efforts de la séparation et évaluation avec des techniques spécifiques dans la résolution des problèmes linéaires de grande taille telles que la génération de colonnes

et la génération de contraintes. Ces méthodes feront l'objet des sections qui suivent.

II.4.3.1 Méthode de "branch-and-price"

La génération de colonnes appliquée à un problème maître en nombres entiers fournit une solution qui n'est pas toujours entière. Pour obtenir une solution entière deux méthodes existent. La première est une approche heuristique, consistant à effectuer une recherche arborescente par séparation et évaluation sur le problème maître obtenu après le processus de génération de colonnes. Cependant, la solution ainsi trouvée peut ne pas être optimale (ni même réalisable) pour le problème initial. En effet, des colonnes, hors la formulation courante, sont susceptibles d'améliorer la solution (ou de rendre le problème faisable) au cours de la recherche arborescente. La deuxième approche est une approche exacte appelée "Branch and Price" dont l'idée de base est de combiner la méthode de génération de colonnes avec la méthode par séparation et évaluation (Branch and Bound). Cette combinaison comprend l'adaptation des principes du branch and bound au contexte de la génération de colonnes en permettant d'expliciter de nouvelles colonnes à chaque nœud de l'arbre, c'est-à-dire en utilisant la méthode de génération de colonnes comme procédure d'évaluation des nœuds.
Soit $\bar{x}$ la solution calculée par la technique de génération de colonnes en un nœud donné. Deux cas se présentent alors :

- $\bar{x}$ est une solution entière, alors le nœud est élagué et $\bar{x}$ sera utilisée comme borne inférieure (problème de maximisation).
- $\bar{x}$ est une solution non entière, alors un branchement sera efféctué.

La recheche arborescente par branch and price permet donc théoriquement de trouver la solution optimale d'un PLNE. Elle peut être également utilisée dans le cadre d'une approche heuristique pour obtenir des solutions de bonne qualité.
La méthode du Branch-and-Price est représentée dans la figure II.3 où les notations suivantes sont utilisées :

- `LB` : la meilleure solution réalisable calculée,
- $\mathtt{x}^*$: une solution optimale du problème (LP),
- `Liste` : liste des problèmes à traiter.

Pour plus d'indications sur la méthode de branch and price, nous renvoyons à (Barnhart et al [6], Vanderbeck & Wolsey [81], Vanderbeck [80]) qui détaillent les différents aspects théoriques du branch and price et discutent en particulier des types de séparation envisageables.

Entrée : Une instance du problème (PLNE) ;
Sortie : Une solution optimale de (PLNE) ;

1. **Initiation**
   - LB $= 0$; $x^* = 0$; Liste $= \emptyset$; initialiser (LP) avec les variables de Ω ;
   - Appliquer l'algorithme GC (figure II.1) au problème LP et obtenir une solution $\bar{x}_{init}$ de valeur Sol_{init} ;
   - **Si** ($\bar{x}_{init}$ est "entière") **alors**
     LB $= Sol_{init}$; $x^* = \bar{x}_{init}$; Aller à 3 ;
   - **Sinon** Ajouter LP dans Liste ;
2. **Tant que** (Liste $\neq \emptyset$) **faire**
   (a) Sélectionner un problème LP dans Liste ;
   (b) Effectuer un branchement et créer deux ou plusieurs sous problèmes de LP ;
   (c) Evaluer les sous problèmes :
   Pour (chaque sous-problème) **faire** :
   i. Appliquer l'algorithme GC au sous-problème et obtenir une solution $\bar{x}$ de valeur $SolCourante$.
   ii. Evaluer la solution $\bar{x}$:
   - **Si** ($\bar{x}$ est "entière") **alors**
     Si ($SolCourante \geq$ LB)
     LB = $SolCourante$; $x^* = \bar{x}$;
   - **Sinon**
     Ajouter le sous-problème dans Liste ;

   Fin Tant que
3. **Fin** : Sortir avec une solution optimale x^* de (PL).

FIG. II.3 : Algorithme de Branch and Price : BP

II.4.3.2 Approche polyèdrale pour les problèmes en nombres entiers

La résolution d'un programme linéaire en nombres entiers (PLNE) par la méthode des plans de coupe, présentée dans le section II.4.2, produit une solution qui est, généralement, fractionnaire bien qu'il n'y ait plus d'inéquation violée dans le système des contraintes du problème. Toutefois, il est possible de résoudre les problèmes PLNE par cette technique

en utilisant des plans de coupe spécifiques à ce type de problèmes. La méthode est appelée dans ce cas approche polyèdrale. Elle consiste à rajouter des contraintes linéaires, appelées coupes, basées sur l'utilisation des informations conçernant l'enveloppe convexe des solutions réalisables. Il s'agit ici de dériver une bonne formulation de l'ensemble des solutions en identifiant des inégalités linéaires pouvant faire partie de l'enveloppe convexe des solutions réalisables. Les premiers travaux qui concernent l'étude polyèdrale ont été réalisés par Dantzig, Fulkerson et Johnson [19] pour résoudre le problème de voyageur de commerce puis ceux de Gomory [32] qui a développé un algorithme de coupes pour les problèmes linéaires en nombres entiers. Par la suite plusieurs travaux se sont succédés dans la même voie comme par exemple ceux de Chvàtal [14] qui a prouvé que toutes les inégalités nécessaires pour décrire l'enveloppe convexe des solutions entières peuvent être obtenues en considérant des combinaisons linéaires des contraintes originales et des contraintes générées et d'appliquer après un schéma d'arrondissement (à condition que les solutions en nombres entiers soient bornées). Ce résultat a été ensuite généralisé par Schrijver [74] aux polyèdres rationnels mais pas nécessairement borné.
Notons, qu'une description totale de l'enveloppe convexe pour un problème NP-difficile est aussi NP-difficile. Pour ce faire, lors de la résolution d'un programme linéaire en nombres entiers par la méthode des plans de coupe, on peut tout d'abord utiliser des plans généraux de coupe de l'enveloppe convexe des solutions du problème. Ces coupes ne sont pas spécifiques au problème résolu, et peuvent être utilisées pour tous les problèmes linéaires en nombres entiers. Les plans de *coupe de Gomory* [32] ou les coupes générées par la méthode dite du *"lift and project"* introduite par Balas, Céria et al ([3], [4] et [5]) en sont des exemples mais en pratique, ce ne sont pas les plus utilisées.
Notons aussi que les méthodes de coupe, utilisées seules, affichent des performances modestes. En revanche, elles se révèlent efficaces associées à des méthodes de recherche arborescente (Branch- and-cut) qu'on décrira plus loin.

II.4.3.3 Quelques notions d'algèbre linéaire et de théorie des polyèdres

Pour la compréhension des méthodes des coupes polyèdrales, les notions suivantes sont indipensables :

1. les vecteurs $x^1, ..., x^k$ de $\mathbb{R}^n$ sont dits linéairement indépendants si l'unique solution

de $\sum_{i=1}^{k} \lambda_i x_i = 0$ est $\lambda = 0$.

2. Les vecteurs de $x^1, ..., x^k$ de $\mathbb{R}^n$ sont dits affinement indépendants si l'unique solution de $\sum_{i=1}^{k} \lambda_i x_i = 0$, $\sum_{i=1}^{k} \lambda_i = 0$ est $\lambda = 0$.
3. l'indépendance linéaire implique l'indépendance affine et non l'inverse.
4. Un polyèdre P est de dimension k si la cardinalité maximale d'un ensemble de vecteurs affinement indépendants de P est $k + 1$. Si P est un polyèdre de $\mathbb{R}^n$ de dimension n, P est dit de pleine dimension.
5. Le rang d'une matrice A est le maximum nombre de lignes (colonnes) linéairement indépendants.
6. La dimension d'un polyèdre P est donnée par la relation :

$$dim(P) = n - rang(A^{=}, b^{=})$$

où $(A^{=}, b^{=})$ correspond aux inégalités vérifiées à l'égalité par tout point de P.

7. Soit $a \in \mathbb{R}^n$, $b \in \mathbb{R}$ tel que $a \neq 0$, l'ensemble $H = \{x \in \mathbb{R}^n : a^T x = b\}$ est appelé hyperplan de $\mathbb{R}^n$.
8. Une inégalité valide est une inégalité satisfaite par tous les points du polyèdre.
9. Soit $\pi x \leq \pi_0$ une inégalité valide pour un polyèdre P, alors

$$F = \{x \in P : \pi x = \pi_0\}$$

est appelé une face de P. Elle est dite propre si $F \neq \emptyset$ et $F \neq P$.

10. Une face propre F de P est dite facette si $dim(F) = dim(P) - 1$.

Il est important de noter que les facettes d'un polytope sont les meilleurs plans de coupe qu'on puisse générer. Cependant, pour des problèmes linéaires en nombres entiers appartenant à la classe des problèmes NP-difficiles, on ne connait pas la description linéaire complète du polytope associé. Et même si on possède une description partielle des facettes du polytope du problème étudié, on ne connait pas d'algorithmes exacts de séparation pour toutes ces familles de facettes (voir par exemple Grötschel, Lovàsz et Schrijver [35]). Ainsi, on utilise généralement des heuristiques de séparation et de ce fait la méthode des plans de coupe s'arrête souvent avant d'avoir trouvé la solution optimale du problème. Pour trouver des solutions optimales on fait appel, en général, à la méthode de branch-and-cut.

II.4.3.4 Méthode de "branch-and-cut"

L'algorithme du "branch-and-cut" est une méthode combinant l'algorithme du "branch-and-bound" avec la méthode des coupes polyèdrales. Ainsi, pour résoudre un programme linéaire en nombres entiers, le "branch-and cut" commence par résoudre une relaxation du problème puis il applique la méthode des coupes polyèdrales sur la solution trouvée. Si celle-ci n'arrive pas à obtenir une solution entière alors le problème est divisé en deux ou plusieurs sous-problèmes qui seront résolus de la même façon. L'algorithme de base est décrit dans la figure II.4 ci-dessous :

Entrée : Une instance du problème (PLNE) ;
Sortie : Une solution optimale de (PLNE) ;

1. **Initiation**
   - LB $= 0$; $x^* = 0$; Liste $= \emptyset$; initialiser le problème LP avec les contraintes de ω ;
   - Appliquer l'algorithme de coupes polyèdrales au problème LP et obtenir une solution $\bar{x}_{init}$ de valeur Sol_{init} ;
   - **Si** ($\bar{x}_{init}$ est "entière") **alors**
     LB = Sol_{init} ; $x^* = \bar{x}_{init}$; Aller à 3 ;
   - **Sinon** Ajouter LP à Liste ;
2. **Tant que** (Liste $\neq \emptyset$) **faire**
   (a) Sélectionner un problème LP dans Liste ;
   (b) Effectuer un branchement et créer deux ou plusieurs sous problèmes de LP ;
   (c) Evaluation des sous problèmes :
   Pour (chaque sous-problème) **faire** :
   i. Appliquer l'algorithme de coupes polyèdrales au sous-problème et obtenir une solution $\bar{x}$ de valeur $SolCourante$.
   ii. Evaluer la solution $\bar{x}$:
   - **Si** ($\bar{x}$ est "entière") **alors**
     Si $SolCourante \geq$ LB
     LB = $SolCourante$; $x^* = \bar{x}$;
   - **Sinon**
     Ajouter le sous-problème dans Liste ;

   Fin Tant que
3. **Fin** : Sortir avec une solution optimale x^* de (PL).

Fig. II.4 : Algorithme de Branch and Cut : BC

Une telle approche a été utilisée pour la première fois pour le problème de "linear

ordering" par Grötschel, Junger et Reinelt [34]. Le terme de "branch-and-cut" était introduit par Padberg et Rinaldi ([64], [65]) pour un algorithme de résolution du problème du voyageur de commerce. Une version généralisée de ce dernier est présentée et parallélisée par Bouzgarrou [9]. La méthode de branch-and-cut a été utilisée avec succès pour la résolution de divers problèmes d'optimisation combinatoire tels que les problèmes de tournées de véhicules (voir, par exemple, Fischetti et al [26]) et les problèmes de satisfiabilité (voir Joy et all [42]).

II.5 conclusion

Nous avons présenté dans ce chapitre différentes notions élémentaires de la programmation linéaire et la programmation linéaire en nombres entiers. Nous avons abordé également les méthodes de résolution des problèmes linéaires et les PLNEs de grande taille. Nous avons donné les schémas généraux des algorithmes basiques dans la résolution de ces problèmes. Les méthodes que nous avons décrites ici, nous seviront dans les chapitres à venir pour résoudre le MMKP.

Chapitre III

Méthodes heuristiques pour le problème du knapsack généralisé à choix multiple

Ce chapitre est consacré à la résolution approchée du problème du knapsack généralisé à choix multiple. Nous proposons à ce propos deux méthodes de résolution. Une première méthode qui s'appuie sur la génération de colonnes et les techniques d'arrondi qui est par la suite combinée avec une méthode d'énumération implicite. Une deuxième méthode, également basée sur la génération de colonnes et employant une technique de branchement local pour une meilleure exploration de l'espace de recherche. Pour les deux approches, plusieurs algorithmes dérivés seront également présentés.([1], [2])

III.1 Introduction

Dans ce chapitre, nous étudions le problème du knapsack généralisé à choix multiple. Nous nous intéressons particulièrement à la résolution approchée. Nous allons développer deux méthodes de résolution. Dans un premier temps, nous présentons une version simplifiée de la première approche. Cette dernière a pour idée principale de faire coopérer

[1] Cette partie fait l'objet d'un document de travail accépté pour publication dans Computational Optimization and Applications sous le titre : "A Column Generation Method for the Multiple-Choice Multi-Dimensional Knapsack Problem" (2008)

[2] Cette partie fait l'objet d'un document de travail accépté pour publication dans International Journal of Operational Research sous le titre : "Cooperative Algorithms for the Multiple-Choice Multi-Dimensional Knapsack Problem" (2008)

une méthode d'arrondi et un algorithme exact. Elle s'appuie sur deux phases :

1. La première phase consiste à appliquer, de manière itérative, la génération de colonnes et arrondir la solution optimale de la relaxation continue obtenue pour fixer une partie des variables.
2. La deuxième phase est appliquée pour résoudre de façon exacte la partie restreinte du problème.

Dans un deuxième temps, nous présentons une version généralisée de la première approche en l'introduisant dans un algorithme de branch-and-bound tronqué. Elle sera appliquée de façon périodique sur un ensemble de nœuds séléctionnés. Ceci offre la possibilité d'une diversification dans la recherche des solutions.

La deuxième approche s'appuie sur les méthodes de recherche par voisinage. Elle combine deux méthodes assez puissantes : (i) la méthode de branchement local et (ii) la méthode de génération de colonnes. La méthode de branchement local est une méthode de recherche par voisinage, proposée par Fischetti et Lodi [25] où les voisinages sont obtenus en introduisant des inégalités linéaires non-valides. Les voisinages induits peuvent être explorés ensuite en utilisant un solveur de programmmes linéaires tel que Cplex.

Conjuguer les efforts des deux méthodes permet d'une part d'explorer efficacement l'espace de recherche. D'autre part, cela permet d'accélérer le processus de recherche.

Dans notre approche, nous utilisons la première heuristique comme boite noire pour explorer les voisinages. Par la suite, nous présentons une version augmentée dont l'idée essentielle est d'imiter une méthode de Branch and Bound (B&B) mais sans garantir l'optimalité des solutions obtenues. Cette version de l'heuristique peut se résumer par les étapes suivantes :

1. Démarrer avec une solution réalisable.
2. Générer un ensemble de nœuds selon les critères de la séparation dans un algorithme de B&B.
3. Appliquer la deuxième heuristique à un ensemble de nœuds élites (par séléction).

Nous présentons une partie expérimentale pour chaque heuristique pour évaluer ses performances.

III.2 Une première méthode approchée

Dans cette partie, nous commençons par décrire la matrice des contraintes associée au MMKP. Puis, nous présentons une technique de génération de colonnes pour résoudre sa relaxation. Cette dernière nous permettera par la suite d'activer et faciliter la recherche des solutions par la méthode que nous proposons. Ensuite, nous présentons comment produire des solutions réalisables à partir de la relaxation continue du problème. Nous présentons, dans ce contexte, trois algorithmes utilisant la génération de colonnes, une technique d'arrondi et éventuellement un algorithme exact tronqué. Enfin, nous présentons une partie expérimentale.

III.2.1 Représentation matricielle du MMKP

Nous rappelons que le problème du knapsack généralisé à choix multiple (MMKP) est défini par un ensemble $S = (S_1, ..., S_i, ..., S_n)$ d'objets divisés en n classes disjointes et un vecteur de capacité $R = (R^1, R^2, ..., R^m)$. Chaque classe $i, i = 1, ..., n$ contient $r_i = |S_i|$ objets.

À chaque objet j de la classe i est associé un profit positif v_{ij} et un vecteur poids $W_{ij} = (w_{ij}^1, ..., w_{ij}^2, ..., w_{ij}^m)$. Le but est d'attribuer au sac, exactement un et un seul objet par classe avec le maximum de profit sans violer les contraintes de capacité.

Formellement le MMKP peut être écrit comme suit :

$$(MMKP)\begin{cases} Z(x) = \max \displaystyle\sum_{i=1}^{n}\sum_{j=1}^{r_i} v_{ij}x_{ij} & \\ \text{s.c.} \displaystyle\sum_{i=1}^{n}\sum_{j=1}^{r_i} w_{ij}^k x_{ij} \leq R^k, & k \in \{1,\dots,m\} \quad (1) \\ \displaystyle\sum_{j=1}^{r_i} x_{ij} = 1, & i \in \{1,\dots,n\} \quad (2) \\ x_{ij} \in \{0,1\}, & i \in \{1,\dots,n\},\ j \in \{1,\dots,r_i\}. \end{cases}$$

ou également sous la forme équivalente suivante :

$$(MMKP)\begin{cases} Z(X) = \max \sum_{i=1}^{n} V_i X_i \\ \text{s.c.} \sum_{i=1}^{n} A_i X_i \leq R^T \\ D_i X_i = d_i, \quad i \in \{1,\ldots,n\} \\ X_i \in \{0,1\}^{r_i}, \quad i \in \{1,\ldots,n\} \end{cases}$$

où :

- $V_i = (v_{i1}, v_{i2}, ..., v_{ir_i}), i = 1, ..., n$: r_i-vecteur ligne des profits correspondant à la classe i.
- $X_i = (x_{i1}, x_{i2}, ..., x_{ir_i}), i = 1, ..., n$: r_i-vecteur colonne des variables de la classe i.
- $A_i = \begin{pmatrix} w_{i1}^1 & w_{i2}^1 & \cdots & w_{ir_i}^1 \\ w_{i1}^2 & w_{i2}^2 & \cdots & w_{ir_i}^2 \\ \vdots & & & \\ w_{i1}^m & w_{i2}^m & \cdots & w_{ir_i}^m \end{pmatrix}, i = 1, ..., n,$

  $(A_i)_{i=1,...,n}$: matrice $(m \times r_i)$ des capacités associées à la classe i.

  Les contraintes formées par les matrices $A_1, A_2, ..., A_n$ sont les contraintes couplantes du système.
- $D_i = (1, 1, ..., 1)$: vecteur ligne de dimension r_i correspondant à la contrainte du choix de la classe i.
- $d_i = 1 \; \forall i \in \{1, \ldots, n\}$.

Les contraintes du problème (MMKP) forment une matrice creuse bloc-angulaire de la forme suivante :

A_1	$\cdots$	A_n
D_1		
	$\ddots$	
		D_n

III.2.2 Une technique de génération de colonnes pour le MMKP

D'un point de vue expérimental, la formulation originale du MMKP n'est pas souhaitable à utiliser ; spécialement, pour des instances du problème de taille large. En effet, le nombre des variables et des contraintes augmente rapidement avec le nombre de classes et le nombre des variables de chacune d'elles, respectivement. De plus, l'utilisation du solveur commercial "Cplex" ne permet de résoudre, de façon optimale (ou proche-optimale), en temps raisonnable, que des instances de petite taille dont le nombre de classes n'excède pas 50 classes de 10 éléments chacune (c-à-d 500 variables au total).
Dans cette partie, nous commençons par présenter une méthode de génération de colonnes pour le MMKP. Ensuite, nous décrivons comment générer une solution initiale et construire le problème maître restreint initial.

III.2.2.1 Adaptation de la génération de colonnes pour le MMKP

Pour appliquer la génération de colonnes au MMKP, nous considérons sa LP-relaxation continue définie comme suit :

$$
\begin{aligned}
Z^{LP}_{MMKP} = \quad & \max \quad \sum_{i=1}^{n}\sum_{j=1}^{r_i} v_{ij}x_{ij} & & \\
& \text{s.t.} \quad \sum_{i=1}^{n}\sum_{j=1}^{r_i} w^k_{ij}x_{ij} \leq R^k, \quad k \in \{1,\ldots,m\} & & (1)\\
& \qquad \sum_{j=1}^{r_i} x_{ij} = 1, & i \in \{1,\ldots,n\} & (2)\\
& \qquad 0 \leq x_{ij} \leq 1, & i \in \{1,\ldots,n\},\ j \in \{1,\ldots,r_i\}. &
\end{aligned}
$$

Nous définissons le problème maître de la génération de colonnes de manière similaire à la LP-relaxation ci-dessus, à l'exception que S sera remplacé par le sous-ensemble S_0 de colonnes (variables) où $S_0 = \cup_{i=1}^{n} S_{i0}$ ($S_{i0} \subseteq S_i,\ S_0 \subseteq S$). Il peut êre décrit de la manière suivante :

$$
\begin{aligned}
Z^{RLP}_{MMKP} = \quad & \max \quad \sum_{i=1}^{n} \sum_{j \in S_{i0}} v_{ij} x_{ij} \\
& \text{s.t.} \quad \sum_{i=1}^{n} \sum_{j \in S_{i0}} w^k_{ij} x_{ij} \leq R^k, \quad k \in \{1, \ldots, m\} \qquad (3) \\
& \qquad \sum_{j \in S_{i0}} x_{ij} = 1, \qquad i \in \{1, \ldots, n\} \qquad (4) \\
& \qquad 0 \leq x_{ij} \leq 1, \qquad i \in \{1, \ldots, n\},\ j \in S_{i0}.
\end{aligned}
$$

Notons ici qu'il est nécessaire d'assurer la faisabilité du problème maître. Pour ce faire, le sous-ensemble des colonnes séléctionnées doit satisfaire les équations (1) et (2). Un choix particulier de l'ensemble S_0 des colonnes initiales correspond à la solution qui sera décrite dans la section III.2.2.2.

Notons aussi que lorsque le problème maître est résolu, nous devons identifier s'il existe encore des colonnes susceptibles d'améliorer la solution en cours et les rajouter à l'ensemble S_0 si c'est le cas. Dans la terminologie de la programmation linéaire et précisément pour le MMKP, ceci correspond à examiner s'il existe une certaine colonne j appartenant à une certaine classe i, et qui ne fait pas partie de la formulation courante du problème, pour laquelle la variable correspondante x_{ij} a un coût réduit strictement positif, noté c_{ij} et défini comme suit :

$$c_{ij} = v_{ij} - \sum_{k=1}^{m} \lambda_k w^k_{ij} - \gamma_i,$$

où $\lambda_k,\ k \in \{1, \ldots, m\}$, et $\gamma_i,\ i \in \{1, \ldots, n\}$, représentent les variables duales associées aux équations (3) et (4), respectivement. Le coût réduit est calculé en résolvant le problème d'optmisation suivant :

$$(SP) \quad \max_{i \in \{1,\ldots,n\}} \Big\{ Z(SP_i) \Big\},$$

où $Z(SP_i)$ représente la solution optimale de (SP_i) définie de la façon suivante :

$$
(SP_i) \begin{cases}
\max & \sum\limits_{j=1}^{r_i} \Big(v_{ij} - \sum\limits_{k=1}^{m} w^k_{ij} \lambda_k \Big) x_{ij} - \gamma_i \\
\text{s.c.} & \sum\limits_{j=1}^{r_i} x_{ij} = 1 \\
& x_{ij} \in \{0, 1\} \quad j = 1, ..., r_i.
\end{cases}
$$

Le problème $SP_i,\ i \in \{1, \ldots, n\}$, est équivalent au problème (SP_i) défini comme suit :

$$\max_{j \in S_i} \Big\{ v_{ij} - \sum_{k=1}^{m} \lambda_k w^k_{ij} - \gamma_i \Big\}.$$

Étant donné que la résolution des sous-problemes $SP_i,\ i = 1, \ldots, n$, n'est pas très coûteuse en matière de temps, nous introduisons à chaque itération de la procédure de génération de colonnes, les n colonnes qui correspondent aux meilleurs coûts réduits de chaque sous-problème SP_i.

III.2.2.2 Solution initiale pour le MMKP

Dans cette partie, nous décrivons la technique employée pour initialiser le problème maître lors de la résolution par génération de colonnes. Nous nous basons essentiellement sur la procédure constructive (CP), qui est une procédure gloutonne, proposée par Hifi et al [38] et présentée dans le premier chapitre. Nous procédons selon les deux étapes suivantes : (i) construire une solution réalisable (ii) initialiser la procédure de génération de colonnes avec cette solution (et d'autres colonnes).
Lorsque nous appliquons la procédure (CP) au MMKP, deux cas se présentent : soit la solution obtenue, qu'on notera F, est réalisable soit elle ne l'est pas. Dans le dernier cas, une procédure constructive complémentaire (CCP) est utilisée (voir chapitre I). Cette dernière utilise une stratégie de permutation 2-opt pour rendre la solution réalisable.
Dans les deux cas, la génération de colonnes utilise les colonnes correspondant à la solution obtenue F. Une colonne par classe $i(i = 1, \ldots, n)$ est rajoutée également, (les colonnes rajoutées doivent êre différentes de celle de F), où chaque colonne est choisie suivant le plus grand rapport pseudo-utilité :

$$\max_{1 \leq j \leq r_i} \frac{v_{ij}}{\sum_{k=1}^{m} w_{ij}^k}.$$

Ceci veut dire que nous favorisons la colonne réalisant la plus grande utilité relativement à toutes les contraintes du knapsack. Notons ici, que cette stratégie reste la meilleure parmi toutes celles que nous avions testées, comme inclure plusieurs colonnes de chaque classe, ou la colonne de chaque classe, réalisant le minimum ou la moyenne du rapport pseudo-utilité.

III.2.3 Solutions entières pour le MMKP

Cette section vise la recherche de solutions réalisables pour le MMKP qui soient de qualité et qui nécéssitent un temps de calcul modeste. Toutes les méthodes que nous allons

présenter utilisent la génération de colonnes pour accélérer le processus de recherche. Nous commençons par présenter une procédure simple d'arrondi puis comment nous l'avons améliorée en la combinant avec un algorithme exact. Enfin, nous donnons une version généralisée des méthodes présentées.

III.2.3.1 Une procédure d'arrondi (PA)

Nous présentons ici, une technique pour obtenir une solution réalisable à partir de la solution optimale de la relaxation du problème MMKP. Cette technique consiste à arrondir la solution fractionnaire de façon itérative en appliquant les deux étapes principales suivantes :

1. Soit LP un problème maître restreint. Nous commençons par résoudre la relaxation de LP et fixer par la suite, les variables entières x_{ij} de la solution ainsi obtenue à leurs valeurs courantes. Notons que fixer une variable à 1, entraine la suppression de toutes les autres variables de la même classe. Ensuite, nous sélectionnons une variable fractionnaire condidate à être arrondie. Notre choix porte sur la variable qui a la plus grande valeur fractionnaire.
2. Le problème réduit qui en résulte après la première étape sera soumis au même traitement. De ce fait, sa relaxation sera résolue par la génération de colonnes et la plus grande variable fractionnaire sera arrondie. Ce processus sera réappliqué jusqu' à l'obtention d'une solution entière.

Nontons qu'à l'étape 2, deux cas peuvent être distingués lorque le processus est fini. Dans le premier cas, la solution atteinte vérifie toutes les contraintes du problème et donc représente une solution réalisable pour le MMKP. Dans le deuxième cas, la solution n'est pas réalisable. Dès lors, nous appliquons la phase *rejet* de la procédure constructive CP, présentée dans le premier chapitre, pour éliminer ou réduire le taux d'infaisabilité de la solution. Nous pouvons également dans le dernier cas tenter de fixer la variable x_{ij} à 0.

Notons aussi qu'une telle approche peut produire des solutions modérées, notamment lorsque les solutions atteintes ne sont pas réalisables. En effet, ce phénomène peut être expliqué par la particularité du MMKP, vu que trouver une solution réalisable pour ce dernier est aussi une tâche difficile. D'un autre côté, fixer une variable à 1 provoque la fixation d'un sous-ensemble de variables (appartenant à la même classe) à 0. Nous pensons

que dans certains cas la fixation employée dans la méthode d'arrondi élimine des classes de façon très agressive.

III.2.3.2 Une procédure d'arrondi hybride (PAH)

Lors de l'application de la génération de colonnes, la phase du "pricing" (résolution des problèmes $SP(i), i = 1, ..., n$) nécessite un temps de calcul négligeable pour enrichir l'ensemble S_0 par de nouvelles colonnes. Nous transférons alors le travail habituellement utilisé dans la phase du "pricing" pour résoudre d'autres sous-problèmes. Ceci est effectué en combinant quelques traitements de la procédure d'arrondi avec un algorithme exact tronqué pour résoudre des sous-problèmes en variables entières réduits. En effet, le processus utilisé peut être vu comme une procédure à deux phases dans laquelle une procédure d'arrondi et un algorithme exact coopérent pour résoudre le MMKP. Dans la première phase, la procédure d'arrondi est utilisée pour fixer une partie des variables et dans la deuxième une méthode exacte spéciale est utilisée pour résoudre le problème réduit qui en résulte.
La méthode exacte considérée est un branch-and-bound restreint dont le nombre de nœuds à explorer est fixé par avance. Nous aurons l'occasion de revenir sur ce point lorsque nous aborderons la partie expérimentale dans la section III.2.4.
La procédure PAH est décrite dans la figure III.1.

III.2.3.3 Une procédure d'arrondi augmentée (PAG/PAHG)

La méthode de séparation et évaluation reste la technique la plus utilisée pour résoudre les problèmes combinatoires (voir chaptre I). Son principe qui consiste à réduire l'espace de recherche des solutions en le divisant en sous-espaces simplifie la recherche et permet même, quelquefois, d'aboutir à de bonnes solutions tout au début de l'exploration des sous-espaces édifiés. Pour tirer profit de ces particularités de la méthode de séparation et évaluation et parvenir à bonifier les méthodes d'arrondi qu'on a présentées précédemment, nous allons décrire dans cette partie une méthode heuristique pour résoudre le problème MMKP. Cette dernière est basée essentiellement sur la génération de colonnes et les méthodes d'énumérations implicites. Elle simule un algorithme exact mais sans garantir l'optimalité des solutions qu'elle fournit. Néanmoins, l'utilisation du

Procédure d'arrondi hybride à deux phases :

Phase1

Soit I_1 et I_2 deux sous-ensembles tels que : $|I_1 \cup I_2| = n$ et $I_1 \cap I_2 = \emptyset$, où I_1 contient les classes déjà fixées (un des objets appartenant à cette classe est fixé à 1) et I_2 est l'ensemble des classes non-fixées (libres).
Les étapes suivantes sont appliquées :

i. Appliquer la procédure d'arrondi pour fixer quelques classes à partir de la solution de la relaxation de LP (construction des ensembles I_1 et I_2).

ii. Appliquer la procédure d'arrondi sur les variables fractionnaires (sur I_2) pour fixer $\alpha\%$ des variables fractionnaires (classes) dans la solution du LP, où $\alpha \in]0,\ 100]$.

Phase2

Appliquer un algorithme exact tronqué sur le reste du problème et sortir avec la solution obtenue.

Fig. III.1 : Procédure d'arrondi hybride : PAH

concept, assez puissant, de la génération de colonnes permet de : (i) accélérer le processus de recherche et (ii) produire des solutions satisfaisantes pour le problème.
La procédure peut se résumer dans les trois étapes principles suivantes :

1. Démarrer le processus de recherche avec une solution initiale (section III.2.2.2)
2. Engendrer un sous ensemble de nœuds dépendant essentiellement des stratégies de séparation employées (voir la section III.2.3.4).
3. Appliquer une méthode heuristique à quelques nœuds sélectionnés.

Dans l'étape 3, l'heuristique utilisée considère deux procédures alternatives : la procédure (PA) et la procédure (PAH). Pour differencier les deux cas, nous notons la procédure d'arrondi augmenté par PAG lorsque PA est utilisée et par PAHG dans le cas contraire. Quelle que soit la procédure adoptée, elle sera appliquée uniquement à quelques nœuds sélectionnés. Ces nœuds peuvent être considérés comme les nœuds élites. Cependant, choisir les meilleurs nœuds représente une tâche compliquée et pas évidente pour le MMKP. Pour cette raison, nous opérons en introduisant deux paramètres empiriques, notés β_1 et β_2.

Le paramètre β_1 représente le nombre de nœuds générés et le paramètre β_2 est introduit pour créer une diversification dans l'espace de recherche. Autrement dit, β_2 est utilisé pour introduire une sorte de résolution périodique par la méthode PAH (ou PA) au lieu de résoudre tous les nœuds générés.
Un schéma général de la procédure d'arrondi augmentée est représenté dans la figure III.2.

Entrée : `Une solution initiale pour le MMKP.`

Sortie : `Une solution réalisable améliorée.`

1. Construire une solution réalisable *initiale* du problème original MMKP et soit ILP le problème linéaire *restreint* initial
2. **Tant que** (condition-d'arrêt-non-vérifiée)
   (a) Soit η un *nœud developpé* (représentant le nœud racine au début) correspondant au problème *restreint* ILP associé au MMKP. Alors,
      i. Appliquer la procédure de *génération de colonnes* au problème relâché ILP correspondant au nœud sélectionné η.
      ii. Appliquer une heuristique *gloutonne d'arrondi* sur une partie des variables de ILP (correspondant à η), notée I_1, et compléter la seconde partie, notée I_2, en utilisant une *procédure spéciale de résolution.*
   (b) Créer de nouveaux nœuds en utilisant différentes *stratégies de branchement*
   (c) supprimer le nœud η ainsi que tous les nœuds dont la valeur objectif est inférieure ou égale à la valeur de la meilleure solution réalisable calculée depuis le début.

   Fin Tant que
3. Sortir avec la meilleure solution .

Fig. III.2 : Procédure d'arrondi augmentée

Notons que l'algorithme s'arrête lorsque le nombre de nœuds limite β_1 est atteint ou lorsque tous les nœuds ont été explorés (la liste des nœuds est vide).

III.2.3.4 Techniques de séparation

Généralement, les méthodes par séparation et évaluation, fondées sur la relaxation continue des programmes linéaires, présument que toutes les solutions fractionnaires peuvent être éliminées par les séparations successives de l'espace potentiel des solutions réalisables.

Une première stratégie de séparation, assez simple et déductive, consiste en la conception d'un ensemble de règles qui permet d'exclure n'importe quelle solution fractionnaire tout en garantissant une séparation valide de l'espace (sous-espace) de recherche. Une deuxième stratégie de séparation, utilisée dans les approches similaires à la nôtre, tient en considération l'exploit réalisé dans l'approche énumérative.

Pour le MMKP, qui est un problème de maximisation, lorsque le processus de génération de colonnes arrive à terme c'est-à-dire, ne parvient plus à détecter des variables avec des coûts réduits positifs, la séparation devient nécéssaire. Nous utilisons trois stratégies de séparation pour résoudre le MMKP.

a) Une première séparation

Cette technique de séparation consiste à réaliser une partition de l'espace de recherche courant (correspondant au nœud sélectionné) en deux parties indépendantes. Tout d'abord, elle commence par choisir une variable fractionnaire de la solution du problème relâché LP. Puis, elle crée les deux branches disjointes suivantes : $x_{ij} = 0$ et $x_{ij} = 1$. Dans notre méthode de résolution, nous choisissons comme variable de séparation, la variable la plus fractionnaire (la variable fractionnaire la plus proche de 1).

b) Une deuxième séparation

L'une des particularités du problème MMKP, concerne la relation étroite qui existe entre les variables appartenant à une même classe. En effet, fixer une variable, entraine la suppression de toutes les autres variables (de la même classe). Plus formellement, la fixation de la variable x_{pj} de la p-ème classe à 1 implique la fixation de $|S_p| - 1$ variables à 0. En exploitant cette particularité du problème, on peut forcer la fixation de plusieurs éléments à zéro dans la première branche et, forcer la fixation d'un élement à 1 dans la seconde. Par conséquent, la première branche correspond à l'ajout de la contrainte associée à la p-ème classe ; définie comme suit :

$$x_{p1} = x_{p2} = \ldots = x_{p\ell} = 0, \tag{III.1}$$

et la deuxième branche correspond à l'injection de la contrainte suivante :

$$\sum_{j=\ell+1}^{|S_p|} x_{pj} = 0, \tag{III.2}$$

où ℓ représente l'indice de la première variable ayant une valeur fractionnaire. Notons que le choix de la première variable fractionnaire preserve le plus grand rapport pseudo-utilité.

Dans nos expérimentations, nous avons également testé d'autres variables fractionnaires comme prendre la dernière par exemple, mais les résultats obtenus ainsi sont très limités. Notre stratégie courante (considérer la première variable fractionnaire) donne des résultats satisfaisants.

c) Une troisième séparation

Les deux stratégies de séparation qu'on vient de décrire, sont plutôt utilisées pour séparer des objets qui font partie d'une même classe. Dans cette partie, nous présentons une autre stratégie qui agit sur des objets appartenant à des classes différentes. Elle peut être considérée comme une séparation de contraintes pour le nœud courant en deux nœuds complémentaires. Notons par x le nœud en cours de traitement et par E_x l'ensemble des indices des dernières colonnes rajoutées au nœud x (nous rappelons ici qu'à chaque itération de la génération de colonne, une colonne est ajoutée par classe).

Nous créons les deux branches suivantes :

$$\sum_{i_j \in E_x} x_{ij} \leq \left\lfloor \frac{|E_x|}{2} \right\rfloor,\ i \in \{1, \ldots, n\},\ j \in \{1, \ldots, r_i\}, \tag{III.3}$$

et

$$\sum_{i_j \in E_x} x_{ij} \geq \left\lceil \frac{|E_x|}{2} \right\rceil,\ i \in \{1, \ldots, n\},\ j \in \{1, \ldots, r_i\}. \tag{III.4}$$

Plusieurs autres techniques de séparations dérivant de celle qu'on vient de présenter peuvent être envisagées. Celles-ci dépendent essentiellement de l'ensemble E_x. En effet, nous pouvons choisir (dans le problème relâché), les éléments fractionnaires du nœud en cours, ou un sous-ensemble d'éléments fixés à 1, ou même combiner les deux.

Gestion des séparations

Dans ce paragraphe, nous proposons un moyen pour organiser le processus de séparation. Il est clair que plusieurs stratégies peuvent être utilisées dans ce contexte. En effet, comme il sera décrit plus loin lorsque nous aborderons la partie expérimentale (section III.2.4), plusieurs tests ont été effetués afin de trouver un bon compromis entre le temps de résolution et la qualité des solution. De façon générale, on peut considérer les séparations présentées dans la section III.2.3.4, combinées ou séparées.

Cependant, nous nous contentons ici de présenter le processus adopté dans nos recherches. Il semble être le plus satisfaisant, vu la qualité des solutions qu'il fournit en temps de calcul raisonnable. Ce dernier est organisé de la façon qui suit :

1. Appliquer la première séparation sur variables, lorsque $|E_x| = 1$.
2. Utiliser la seconde séparation, lorsque $|E_x| \geq 2$.
3. Appliquer la séparation complémentaire (séparation 3), si c'est la deuxième séparation qui était appliquée durant la dernière itération et le nombre des colonnes introduites est au moins égal à 2.

III.2.4 Partie expérimentale

Dans cette section, nous allons exposer les résultats numériques obtenus par les deux versions de l'approche d'arrondi augmentée (PAG et PAHG). L'objectif de ces expériences est double : (i) d'une part, déterminer un meilleur compromis entre les solutions obtenues et le temps d'exécution et (ii) évaluer les performances des deux versions de l'approche en comparant les résultats obtenus, d'un côté, aux résultats de la recherche locale réactive, notée MRLS, proposée par Hifi et al [39]. D'un autre côté, nous les comparons aux résultats fournis par le solveur commercial Cplex v.9 après une heure de temps de résolution. Les méthodes PAG et PAHG ont été codées en **C++** et exécutées sur une station SUN Ultra-Sparc10 (250 Mhz et avec 1Gb de mémoire RAM).
Pour évaluer les performances de PAG et PAHG, nous utilisons 33 instances, représentées dans la Table III.1, qui correspondent à deux groupes. Le premier groupe contient 13 instances, notées `I1,...,I13` dans le Tableau III.1, utilisées par Khan et al [47]. Le deuxième groupe contient 20 instances, notées `Ins01,..., Ins20`, utilisées par Hifi et al [39] et variant de taille moyenne à taille large. Pour chaque instance, nous reportons le nombre n de classes, le nombre r_i d'éléments dans chaque classe `i, i = 1,..., n`, le nombre de contraintes de ressources m et finalement le nombre total de variables $\sum_{i=1}^{n} r_i$.

III.2.4.1 Performances de la méthode PAG

En général, la résolution des problèmes d'optimisation combinatoire par des méthodes approchées implique l'utilisation de plusieurs paramètres qui influencent la qualité des solutions obtenues ainsi que le temps de résolution. En effet, un bon ajustement des paramètres mène souvent à des solutions meilleures mais parfois plus coûteuses en terme du temps de calcul. Les valeurs choisies dans nos expérimentations représentent un bon compromis entre la qualité des solutions et le temps de résolution nécessaire.

Groupe 1					Groupe 2				
#Inst.	n	r_i	m	$\sum_{i=1}^{n} r_i$	#Inst.	n	r_i	m	$\sum_{i=1}^{n} r_i$
I01	5	5	5	25	Ins01	50	10	10	500
I02	10	5	5	50	Ins02	50	10	10	500
I03	15	10	10	150	Ins03	60	10	10	600
I04	20	10	10	200	Ins04	70	10	10	700
I05	25	10	10	250	Ins05	75	10	10	750
I06	30	10	10	300	Ins06	75	10	10	750
I07	100	10	10	1000	Ins07	80	10	10	800
I08	150	10	10	1500	Ins08	80	10	10	800
I09	200	10	10	2000	Ins09	80	10	10	800
I10	250	10	10	2500	Ins10	90	10	10	900
I11	300	10	10	3000	Ins11	90	10	10	900
I12	350	10	10	3500	Ins12	100	10	10	1000
I13	400	10	10	4000	Ins13	100	30	10	3000
					Ins14	150	30	10	4500
					Ins15	180	30	10	5400
					Ins16	200	30	10	6000
					Ins17	250	30	10	7500
					Ins18	280	20	10	5600
					Ins19	300	20	10	6000
					Ins20	350	20	10	7000

Tab. III.1 : Détails des instances

Les deux méthodes PAG et PAHG utilisent plusieurs paramètres intrinsèques tels que le nombre d'arborescences β_1 (nœuds élites) à traiter et le paramètre de fréquence β_2 introduit pour étendre les ségments du domaine de recherche à explorer.

Pour définir la valeur du nombre maximum de nœuds à résoudre par l'algorithme, nous varions le paramètre β_1 dans l'intervalle discret $\{200, 500, 1000\}$. Les tests sont effectués en fixant également le paramètre de fréquence β_2 dans l'intervalle discret $\{5, 10, 25\}$. Ce dernier représente le nombre des séparations utilisées avant de développer le dernier nœud élite généré. D'autres valeurs ont aussi été considérées, mais nous ne reportons ici que les plus significatives d'entre elles. Notons que générer β_1 nœuds signifie que $\beta_1/2$ nœuds peuvent être traités par la procédure PAG vu que chaque stratégie de séparation crée deux nouveaux nœuds potentiels.

Dans le tableau III.2, nous représentons les résultats de la méthode PAG lorsque nous employons les différentes stratégies de séparation (présentées dans la section III.2.3.4), séparées ou combinées.

Pour chaque bloc-ligne, la première colonne correspond à la moyenne du temps d'exécution nécéssaire. La deuxième colonne correspond au nombre de solutions améliorées obtenues par une technique de séparation donnée et lorsque les deux paramètres β_1 et β_2 varient dans les intervalles discrets $\{200, 500, 1000\}$ et $\{5, 10, 25\}$, respectivement. Notons ici que les solutions optimales des six premières instances, sont atteintes par toutes les versions de PAG.

À partir du tableau III.2, on peut remarquer que :

- la méthode PAG fournit des solutions modérées lorsque $\beta_2 = 25$ même si le temps de résolution reste minimal. En effet, PAG fournit entre 1 et 8 meilleures solutions en une moyenne de temps variant entre 5.78 et 29.94 secondes.
- Lorsque $\beta_2 \in \{5, 10\}$ et en variant β_1 dans $\{500, 1000\}$, PAG a un meilleur comportement. Le nombre de meilleures solutions produites dans ce cas varie entre 4 et 17 (sur 27 instances) et le temps de résolution, évalué en moyenne, est compris entre 22.57 et 126.07 secondes.
- La méthode PAG combinant les trois stratégies de séparation peut être considérée comme la meilleure version de PAG vu que le nombre de meilleures solutions générées dans ce cas est plus important. En effet, dans chaque bloc-colonne caractérisant la variation de β_1, on remarque d'une part une augmentation du nombre de solutions améliorées avec à peu près le même temps de calcul en moyenne. D'autre part, le dernier bloc-ligne (dernière ligne, colonne 7) contient le nombre maximum de meilleures solutions obtenues en fixant le paramètre β_1 à 1000, mais nécéssitant un temps important de résolution (126.07).
- De plus, le rapport $\frac{M.CPU}{\#Meil}$ est plus avantageux pour l'ajustement $\beta_1 = 500$ et $\beta_2 = 10$. Il réalise la valeur moyenne de 1.52 comparée à la valeur moyenne de 7.42 représentée par l'ajustement $\beta_1 = 1000$ et $\beta_2 = 25$.

La figure III.3 montre le comportement de PAG lorsqu'on fait varier les stratégies de séparation ainsi que les deux paramètres β_1 et β_2. Trois ensembles sont représentés : un premier ensemble correspondant à $\beta_1 = 200$, noté Set 1, un second ensemble (noté Set 2) correspondant à $\beta_1 = 500$ et un dernier ensemble où $\beta_1 = 1000$, noté Set 3. Les ensembles

		$\beta_2 = 25$		$\beta_2 = 10$		$\beta_2 = 5$	
	β_1	M. cpu	# Meil	M. cpu	# Meil	M. cpu	# Meil
Séparation 1							
	200	7.10	1	18.87	3	27.34	5
	500	13.71	3	33.17	4	49.59	8
	1000	22.56	5	52.40	10	107.13	12
Séparation 2							
	200	5.78	2	14.50	2	20.89	5
	500	9.10	3	22.57	6	38.82	10
	1000	20.13	6	52.22	9	100.17	12
Séparation 3							
	200	9.49	2	22.03	5	30.33	5
	500	14.46	4	37.97	8	52.53	8
	1000	27.20	5	69.47	10	106.72	10
Séparations 1 & 2							
	200	9.23	2	20.44	6	25.52	6
	500	14.53	4	32.16	7	40.75	7
	1000	20.48	5	50.01	13	104.93	13
Séparations 1 & 3							
	200	9.99	2	25.39	4	30.92	4
	500	14.93	3	37.87	7	43.85	7
	1000	21.29	3	57.94	10	110.44	10
Séparations 2 & 3							
	200	7.70	2	18.24	7	24.03	7
	500	9.46	4	24.09	12	43.89	12
	1000	19.56	7	52.31	13	109.56	13
Séparations 1, 2 & 3							
	200	7.59	2	14.23	3	23.63	7
	500	11.62	5	22.79	15	45.18	15
	1000	29.94	8	61.71	15	126.07	17

TAB. III.2 : Comportement de PAG selon la variation des paramètres : nombre de nœuds, fréquence et stratégie de séparation

Set 1, Set 2 et Set 3 reportent les résultats des trois bloc-colonnes du tableau III.2 allant du premier bloc-colonne ($\beta_2 = 25$) au troisième bloc-colonne ($\beta_2 = 5$).
De la figure III.3, on remarque que :

1 La méthode PAG est rapide pour $\beta_1 = 200$ (voir curve 2, Set 1) mais n'atteint qu'un nombre minimum seulement de meilleures solutions (voir curve 1, Set 1)). Dans ce cas, la valeur la plus élevée 7, représentant le nombre maximum de meilleures solutions atteintes, correspond à la valeur $\beta_2 = 5$ qui utilise les stratégies de séparation 2 & 3 ainsi que les stratégies de séparation 1, 2 & 3 et $\beta_2 = 10$, qui utilise les

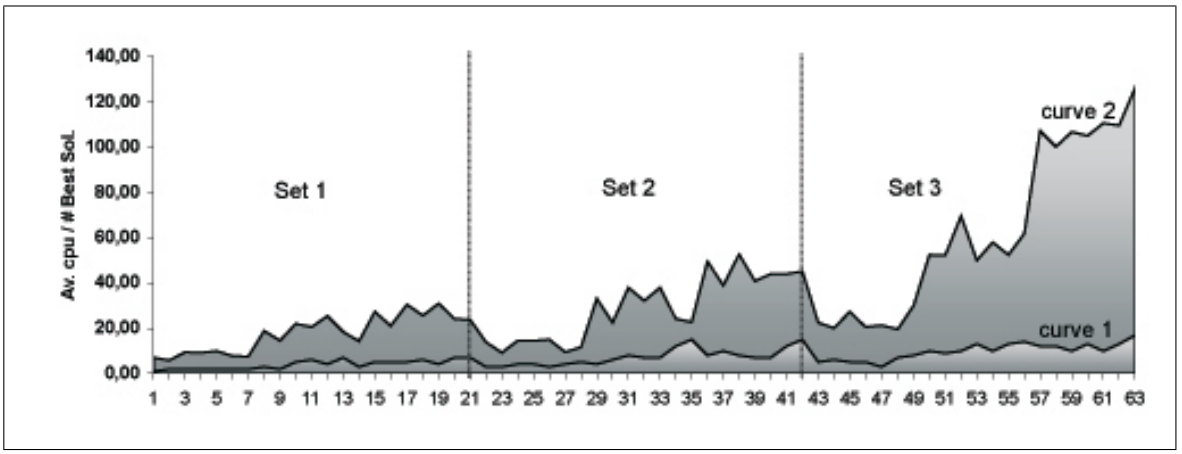

Fig. III.3 : Variations du nombre de meilleures solutions obtenues & le temps moyen de résolution correspondant

stratégies de séparation 2 & 3.

2 Pour $\beta_1 = 500$ et selon la première courbe (curve 1), on peut distinguer deux réglages intéressants concernant la valeur 15 (nombre de meilleures solutions atteintes) : (i) le premier réglage correspond à la valeur 35 dans l'axe des abscisses, qui représente aussi le couple (M.CPU, # Meil) = (22.79, 15) et (ii) le deuxième est dans la position 42 dans l'axe des abscisses, associée au couple (M.CPU, # Meil) = (45.18, 15). Les deux valeurs correspondent à $\beta_2 = 10$ et $\beta_2 = 5$ respectivement.

3 Dans certains cas, la procédure PAG est capable d'atteindre 17 meilleures solutions (voir curve 1, Set 3), mais en consommant plus de temps (voir curbe 2, Set 3).

On peut conclure qu'il n'est pas nécessaire d'augmenter la valeur du paramètre β_1 pour obtenir de meilleures solutions surtout que le temps de résolution dans ce cas croit remarquablement. Dans ce qui suit, nous maintenons les valeurs correspondant au couple $(\beta_1, \beta_2) = (500, 10)$ qui permet de réaliser un bon compromis entre le nombre des meilleures solutions obtenues et le temps moyen de résolution.

Le tableau III.3 évalue les performances de la méthode PAG en comparant ses résultats avec ceux fournis par le solveur Cplex et la méthode MRLS, proposée par Hifi et al [39]. Dans la colonne 2, sont représentées les meilleures solutions (ou les solutions optimales ; dans ce cas, l'instance est marquée par le signe "○") de l'instance. La colonne 3 contient les meilleures solutions obtenues par le solveur Cplex, notées $Cplex_{Mei}$. La colonne 4 reporte les solutions données par MRLS et la colonne 5, le temps d'exécution de cette dernière. Finalement, dans la colonne 6, nous affichons les résultats de la méthode PAG et dans

la colonne 7, le temps de résolution nécessaire pour atteindre ces solutions. Les solutions représentées en caractères italiques dans la colonne 6 indiquent que PAG améliore les meilleures solutions obtenues par le solveur Cplex et la méthode MRLS.
Du tableau III.3, on remarque que :

1. Pour le premier groupe d'instances, la méthode PAG atteint les solutions correspondant aux six premières instances dont la solution optimale est connue (pour ces instances, tous les algorithmes fournissent les solutions optimales). Elle atteint également une (respectivement. trois) solution(s) fournie(s) par MRLS (resp. Cplex) et produit deux nouvelles meilleures solutions parmi les six solutions qui restent. En revanche, PAG échoue dans 3 (resp. une) occasions à atteindre les solutions produites par MRLS (resp. Cplex).
2. Pour le deuxième groupe d'instances, PAG améliore 9 solutions parmi 20, atteint 2 solutions de MRLS et échoue dans 9 (resp. 6) occasions d'atteindre les solutions produites par MRLS (resp. Cplex). Notons aussi que PAG échoue dans 3 cas sur 20 à produire une solution réalisable. Dans ce cas, comme nous avons mentionné dans la section III.2.3.1, nous tentons de construire une solution réalisable en appliquant la phase *rejet* de la méthode CP de Hifi et al [38] aux deux derniers nœuds. La solution obtenue ainsi est marquée dans le tableau III.3 par le symbole "$\star$".
3. MRLS (resp. Cplex) nécéssite 49.19 secondes (resp. une heure) alors que PAG fournit les solutions représentées dans le tableau III.3 en 22.79 secondes, en moyenne.

III.2.4.2 Performances de la méthode PAHG

Dans cette section, nous évaluons les performances de la méthode PAHG. Pour ce faire, nous comparons les résultats de cette dernière avec ceux fournis par MRLS et le solveur Cplex sur les deux groupes d'instances. Cette comparaison est effectuée en limitant le temps de résolution du solveur Cplex à 3600 sec. Les résultats de la méthode MRLS proviennent de Hifi et al [39].
La méthode PAHG est testée en considérant différents ajustements. Ici, nous maintenons les valeurs de β_1 et β_2 utilisées par PAG (voir la section III.2.4.1). Nous rappellons que, d'un côté, PAHG considère deux sous ensembles I_1 et I_2 (le premier sous-ensemble I_1

		Cplex solver	MRLS		PAG	
Inst	Best/Opt Sol	Cplex_{IFS}	Sol.	cpu	Sol.	cpu
I01	173°	173	173	0.59	173	0.16
I02	364°	364	364	0.81	364	0.73
I03	1602°	1602	1602	2.01	1602	1.19
I04	3597°	3597	3597	2.30	3597	2.21
I05	3905.7°	3905.7	3905.7	1.94	3905.7	0.05
I06	4799.3°	4799.3	4799.3	2.37	4799.3	1.55
I07	24587	24584	24587	36.58	24587	11.17
I08	36877	36869	36877	37.00	36869	19.80
I09	49167	49155	49167	25.10	*49175*	18.92
I10	61446	61446	61437	47.00	*61461*	28.87
I11	73773	73759	73773	41.45	73759	40.59
I12	86071	86071	86069	42.08	86071	48.07
I13	98429	98418	98429	160.41	98409	58.85
Ins01	10714	10709	17014	10.27	*10719*	6.21
Ins02	13598	13597	13598	76.00	13460	4.76
Ins03	10943	10934	10943	58.00	10939*	4.97
Ins04	14429	14422	14429	7.69	*14436*	8.15
Ins05	17053	17041	17053	42.00	17047*	9.70
Ins06	16823	16815	16823	50.00	16823	8.87
Ins07	16423	16407	16423	65.00	16310	11.42
Ins08	17506	17484	17506	26.78	*17510*	9.57
Ins09	17754	17747	17754	51.23	*17760*	8.94
Ins10	19314	19285	19314	32.16	19306*	11.43
Ins11	19431	19424	19431	110.98	19431	9.53
Ins12	21730	21725	21730	23.39	21646	14.32
Ins13	21569	21569	21569	18.00	21550	18.89
Ins14	32869	32866	32869	72.00	*32870*	25.79
Ins15	39154	39154	39148	63.00	*39157*	37.53
Ins16	43357	43357	43354	194.00	*43361*	40.69
Ins17	54349	54349	54349	30.00	54329	55.67
Ins18	60456	60455	60456	201.00	*60457*	65.77
Ins19	64921	64919	64921	45.00	64913	74.72
Ins20	75603	75603	75603	47.00	*75610*	92.99
Av. cpu				49.19		22.79

Tab. III.3 : Performance de la procédure d'arrondi PAG

contient les éléments dont les valeurs sont entières, le second contient le reste des variables libres). D'un autre côté, d'autres paramètres supplémentaires et stratégies sont considérés : (i) α_1 (pourcentage global des classes à fixer à leurs valeurs entières), (ii) α_2 (pourcentage des variables à fixer à leurs valeurs entières dans I_1) et (iii) la stratégie de séparation utilisée par l'algorithme. Notons que les deux paramètres α_1 et α_2 sont introduits pour

restreindre le pourcentage des variables à fixer à leurs valeurs entières. Notons aussi que le pourcentage des variables fractionnaires à arrondir, noté α, est remplacé par la variable la plus fractionnaire.

Dans un premier temps, nous analysons le comportement de PAHG lorsque nous variions les deux paramètres α_1 et α_2. Les résultats représentés dans le tableau III.4 correspondent à l'utilisation de la stratégie de séparation décrite dans la section III.2.3.4. Le tableau III.4 reporte le nombre de meilleures solutions fournies (atteintes) quand les deux paramètres α_1 et α_2 sont variés dans l'intervalle discret {25%, 50%, 75%} ainsi que le temps de résolution moyen nécessaire pour chaque version de l'algorithme. Le premier bloc-ligne du tableau représente les résultats obtenus lorsque α_2 est fixé à 75%.
Le deuxième et le troisième blocs-lignes concernent les résultats obtenus lorsque $\alpha_2 = 50\%$ et $\alpha_2 = 25\%$, respectivement. Finalement, notons que PAHG nécessite de limiter, dans certains cas, le nombre de nœuds générés lorsque le solveur Cplex est appliqué aux variables libres qui restent. Dans le but de maintenir le même degré de comparaison, particulièrement pour le temps moyen de résolution, nous considérons un maximum de 5000, 4000 et 3000 nœuds par arbre pour $\alpha_1 = 25\%, 50\%$ et 75%, respectivement. Du tableau III.4, nous remarquons que :

- Lorsque $\alpha_1 = \alpha_2 = 75\%$, PAHG atteint 15 meilleures solutions parmi 33. Cette version peut être considérée comme la version la moins coûteuse de l'algorithme. En effet, le temps moyen de résolution consommé dans ce cas est de 92.57 secondes.
- Quatre versions de l'algorithme, fournissent approximativement le même nombre de meilleures solutions (entre 28 et 30). En effet, quatre couples de (α_1, α_2)-correspondant à (50%, 50%), (50%, 25%), (25%, 50%) et (25%, 25%)-réalisent 30, 29, 28 et 28 meilleures solutions, respectivement.
- Parmi les quatres versions atteingnant les solutions, nous distinguons deux versions plus au moins rapides. La première version correspond au couple (50%, 50%). Elle nécessite 150.99 sec de temps de résolution en moyenne. La seconde version, représente le couple (25%, 50%) et nécessite (158.48 sec), en moyenne.

Du tableau III.4, nous pouvons conclure que choisir des valeurs intermédiaires du couple (α_1, α_2) maintient le pourcentage élevé de meilleures solutions atteintes dans un temps moyen de résolution raisonable.

	$\alpha_1 = 75\%$		$\alpha_1 = 50\%$		$\alpha_1 = 25\%$	
	Av. cpu	Nb. Best/Opt	Av. cpu	Nb. Best/Opt	Av. cpu	Nb. Best/Opt
$\alpha_2 = 75\%$						
Premier groupe	72,03	7	83,00	7	87,87	10
Deuxième groupe	105.92	8	130.14	11	156.96	12
Les deux groupes	92.57	15	111.57	18	129.74	22
$\alpha_2 = 50\%$						
Premier groupe	79.88	9	93.40	12	111.34	12
Deuxième groupe	114.45	10	188.43	18	240.96	17
Les deux groupes	100.83	19	150.99	30	189.90	29
$\alpha_2 = 25\%$						
Premier groupe	107.98	8	125.52	11	150.83	12
Deuxième groupe	138.68	11	179.90	17	280.12	16
Les deux groupes	126.59	19	158.48	28	229.19	28

Tab. III.4 : Comportement de PAHG selon la variation du paramètre α_1

Dans un deuxième temps, pour analyser le comportement de PAHG lorsque différentes stratégies de séparation sont utilisées, nous considérons sept versions de l'algorithme. Trois versions de PAHG utilisent une seule stratégie de séparation (soit la première, la deuxième ou la troisième). Trois autres versions combinent deux différentes séparations et la dernière utilise la version alternative décrite dans la section III.2.3.4. Pour effectuer une comparaison plus complète entre les sept versions de l'algorithme, nous fixons α_2 à 50% (vu le comportement positif de l'algorithme pour cette valeur -voir le deuxième bloc- ligne du tableau III.4). Nous varions également le paramètre α_1 dans l'intervalle discret {25%, 50%, 75%}.

Les différents résultats obtenus avec les différents ajustements sont reportés dans le tableau III.5).

Du tableau III.5, on peut remarquer que :

- Globalement, le pourcentage des meilleures solutions produites par PAHG varie de 42.42% à 90.91% et le temps moyen de résolution varie de 73.85 à 244.98 secondes.
- Lorsque la première stratégie de séparation est utilisée, l'algorithme PAHG possède un bon comportement lorsque α_1 est fixée à 25%. En effet, il atteint dans ce cas 25 meilleures solutions parmi 33 en un temps de résolution légèrement plus élevé (186.73 secondes).
- Les résultats obtenus pour la deuxième stratégie de séparation sont très intéressants, comme mentionné dans le deuxième bloc-ligne. Pour $\alpha_1 = 75\%$, l'algorithme atteint

	$\alpha_1 = 75\%$		$\alpha_1 = 50\%$		$\alpha_1 = 25\%$	
	Av. cpu	Nb. Meil/Opt	Av. cpu	Nb. Meil/Opt	Av. cpu	Nb. Meil/Opt
Séparation 1						
Premier groupe	59 .26	7	72 .31	8	90 .07	10
Deuxième groupe	187 .40	7	229 .85	14	249 .56	15
Les deux groupes	136 .92	14	167 .79	22	186 .73	25
Séparation 2						
Premier groupe	50 .06	8	58 .91	10	69 .75	11
Deuxième groupe	89 .31	9	117 .44	14	162 .52	16
Les deux groupes	73 .85	17	94 .38	24	125 .97	27
Séparation 3						
Premier groupe	101 .66	8	138 .49	9	159 .32	9
Deuxième groupe	181 .44	7	232 .74	12	300 .67	16
Les deux groupes	150 .01	15	195 .61	21	244 .98	25
Séparations 1 & 2						
Premier groupe	89 .43	8	101 .31	11	144 .18	10
Deuxième groupe	140 .51	10	186 .09	13	283 .09	16
Les deux groupes	120 .39	18	151 .65	24	226 .66	26
Séparations 1 & 3						
Premier groupe	79 .84	9	997 .29	12	139 .11	11
Deuxième groupe	185 .34	8	228 .61	10	274 .72	14
Les deux groupes	143 .78	17	176 .88	22	221 .30	25
Séparations 2 & 3						
Premier groupe	90 .20	9	121 .56	10	156 .82	12
Deuxième groupe	148 .37	10	197 .44	15	279 .55	15
Les deux groupes	125 .46	19	167 .55	25	231 .20	27
Séparations 1, 2 & 3						
Premier groupe	79 .88	9	93 .40	12	111 .34	12
Deuxième groupe	114 .45	10	188 .43	18	240 .96	17
Les deux groupes	100 .83	19	150 .99	30	189 .90	29

Tab. III.5 : Comportement de PAHG selon la variation des strategies de séparation

17 meilleures solutions en consommant 73.85 secondes dans la moyenne. Le nombre de meilleures solutions atteintes augmente à 24 lorsque $\alpha_1 = 50\%$ en consommant un peu plus de temps (94.38 secondes). Lorsque $\alpha_1 = 25\%$, PAHG est capable de maintenir un temps de résolution intéressant (125.97 secondes) tout en augmentant le nombre de meilleures solutions atteintes de trois unités.

- D'après les résultats représentés dans le troisième bloc-ligne, l'algorithme utilisant la troisième séparation a le même comportement que lorsque la première séparation est utilisée.
- En se basant sur les premiers résultats de l'algorithme, nous nous limitons prin-

cipalement à la combinaison de la seconde stratégie de séparation avec les autres (première et troisième séparation). D'après les résultats représentés dans le quatrième et le sixième bloc-ligne, respectivement, on peut observer que combiner la première (troisième) séparation avec la deuxième est positif pour l'approche vu le nombre de solutions meilleures/optimales obtenues. En revanche, dans certains cas le temps moyen de résolution croit considérablement : il varie de 120.39 à 226.66 lorsque les deux premières séparations sont combinées et de 125.46 à 231.20 dans le cas où les deux dernières séparations sont utilisées.

- Les résultats de l'algorithme sont plus intéressants quand les trois stratégies de séparation sont combinées suivant le schéma décrit dans la section III.2.3.4. On peut observer à partir du dernier bloc-ligne que :

  1. PAHG maintient le plus grand nombre de solutions meilleures/optimales atteintes dans la première colonne lorsque $\alpha_1 = 75\%$.
  2. En fixant α_1 à 50%, PAHG produit le nombre le plus élevé de solutions meilleures/optimales en consommant 150.99 secondes en moyenne.
  3. Lorsque $\alpha_1 = 25\%$, PAHG maintient un pourcentage élevé des solutions meilleures/optimales produites mais nécessite plus de temps de résolution (189.90 secondes).

La méthode PAHG utilisant les trois stratégies de séparation peut être considérée comme la meilleure version de l'algorithme en voyant le temps moyen de résolution ainsi que la qualité des solutions obtenues.

Dans le tableau III.6, nous représentons pour les deux groupes d'instances les résultats de PAHG lorsque : (i) seulement la deuxième stratégie de séparation est employée avec $\alpha_1 = 25\%$, et (ii) les trois stratégies de séparation sont combinées en variant α_1 dans {25, 50, 75}. Nous reportons également pour les deux groupes, les résultats de MRLS et le solveur Cplex. La colonne 2 représente la meilleure solution ou la solution optimale, d'une instance, produite par un des algorithmes considérés. Lorsque la solution est optimale, l'instance est marquée par le signe "$\circ$". Les colonnes 3, 4 et 5, contiennent les solutions fournies par le solveur Cplex, MRLS et PAG respectivement. Dans les colonnes 6, 8, 10 et 12 nous reportons les solutions des quatre versions de PAHG tandis que dans les colonnes 7, 9, 11 et 13 nous représentons le temps de résolution nécessaire pour chacune d'elles. Notons

que le signe "-" indique si les algorithmes atteignent les meilleures/optimales solutions de l'instance considérée.

Les résultats du tableau III.6 peuvent être analysés comme suit :

- Globalement, chaque version de PAHG atteint des solutions meilleures que le solveur Cplex, MRLS et PAG.
- Si nous considérons comme solution de référence la meilleure solution produite par le solveur Cplex, MRLS et PAG, alors la version de PAHG la plus rapide -utilisant les trois stratégies de séparation- et fixant α_1 à 75% (colonnes 8 et 9) produit 8 meilleures solutions parmi 33. Elle atteint 22 solutions et échoue dans trois occasions à atteindre la solution référence.
- En utilisant la solution référence, on peut observer que la méthode PAHG utilisant la seconde séparation reste compétitive. En effet, cette dernière produit 13 meilleures solutions en consommant 125.97 secondes, en moyenne. Elle atteint 19 solutions et échoue en une seule occasion à atteindre la solution référence. Cette version peut être considérée comme une version intermédiare intéressante pour résoudre le MMKP.
- Les deux dernières versions de PAHG fournissent de meilleurs résulats (colonnes 10-13). En effet, la version où $\alpha_1 = 25\%$ fournit 13 meilleures solutions et atteint 20 solutions références. Quant à la version où $\alpha_1 = 50\%$, PAHG améliore 15 solutions et atteint 18 solutions références.
- Notons aussi qu'avec $\alpha_1 = 25\%$ et $\alpha_1 = 75\%$, PAHG améliore respectivement 20 et 21 solutions (parmi 27) que produisent MRLS et le solveur Cplex. Evidemment, ces résultats sont obtenus en consommant un temps important comparé à celui que nécessite la méthode MRLS. Cependant, l'amélioration de la qualité des solutions justifie le temps additionnel qui reste raisonable.

					PAHG : Procédure d'arrondi hybride augmentée							
		Cplex solver	MRLS	PAG	Séparation 2		Séparations 1, 2 & 3					
					$\alpha_1 = 25\%$		$\alpha_1 = 75\%$		$\alpha_1 = 50\%$		$\alpha_1 = 25\%$	
Inst	Meil/Opt	Cplex_{IFS}	Sol.	Sol.	Sol.	cpu	Sol.	cpu	Sol.	cpu	Sol.	cpu
I01	173	-	-	-	-	0.28	-	0.21	-	0.23	-	0.27
I02	364	-	-	-	-	1.04	-	1.87	-	2.33	-	2.31
I03	1602	-	-	-	-	5.67	-	3.48	-	3.52	-	4.58
I04	3597	-	-	-	-	10.69	-	6.42	-	9.78	-	13.81
I05	3905.70	-	-	-	-	0.09	-	0.07	-	0.08	-	0.08
I06	4799.30	-	-	-	-	0.96	-	4.79	-	4.36	-	4.38
I07	24587	24584	-	-	-	95.83	24584	34.09	-	63.95	-	74.71
I08	36892	36869	36877	36869	36888	138.5	36887	94.35	36890	127.08	-	170.06
I09	49176	49155	49167	49175	-	72.18	49175	72.93	-	96.20	49175	108.24
I10	61461	61446	61437	-	-	98.61	-	119.38	-	151.49	-	188.73
I11	73775	73759	73773	73759	-	103.92	-	146.16	-	169.72	-	194.35
I12	86078	86071	86069	86071	-	211.63	86071	219.96	-	229.26	-	264.95
I13	98431	98418	98429	98409	98429	167.32	-	334.72	-	356.24	-	421.01
Ins01	10732	10709	17014	10719	-	15.65	10719	16.06	-	46.26	10730	57.14
Ins02	13598	13597	-	13460	13597	22.16	-	15.50	-	12.78	-	20.85
Ins03	10943	10934	-	-	-	80.57	-	11.60	-	48.37	-	58.36
Ins04	14440	14422	14429	14436	-	85.96	14438	27.53	-	62.53	-	66.38
Ins05	17053	17041	-	-	-	87.32	-	55.92	-	91.16	-	119.86
Ins06	16825	16815	16823	16823	-	35.91	16823	34.18	16824	75.62	-	108.88
Ins07	16435	16407	16423	16310	16432	76.5	16424	39.69	-	122.40	16432	126.17
Ins08	17510	17484	17506	-	-	37.21	-	36.33	-	74.68	-	83.08
Ins09	17760	17747	17754	-	-	112.56	-	29.03	-	48.23	-	56.11
Ins10	19314	19285	-	19306	-	65.84	19312	68.13	-	143.45	-	188.64
Ins11	19434	19424	19431	19431	-	46.25	19432	33.49	-	56.55	-	170.48
Ins12	21731	21725	21730	21646	21730	229.97	21728	157.62	-	126.58	21730	227.91
Ins13	21575	21569	21569	21550	21571	89.65	21570	153.25	-	198.53	-	184.95
Ins14	32870	32866	32869	-	-	71.16	-	81.75	-	151.51	-	197.28
Ins15	39157	39154	39148	-	-	135.62	-	154.00	-	306.27	-	397.94
Ins16	43361	43357	43354	-	-	320.19	-	162.39	-	263.90	-	328.43
Ins17	54349	-	-	54329	-	386.72	-	212.15	-	269.07	-	313.53
Ins18	60460	60455	60456	60457	-	227.33	60457	309.93	60459	654.59	-	907.84
Ins19	64923	64919	64921	64913	-	450.58	64921	342.33	-	461.14	-	532.47
Ins20	75611	75603	75603	75610	-	673.25	-	348.18	-	554.88	-	672.98
Av. cpu						125.97		100.83		150.99		189.90
Nb Best		6	12	15	27		19		30		29	

Tab. III.6 : Performance de PAHG sur deux groupes d'instances

III.3 Une deuxième méthode approchée

Dans cette partie, nous nous intéressons à la résolution approchée du problème MMKP par les méthodes de recherche par voisinage. Nous allons présenter trois versions d'un algorithme approché. La première, est une adaptation de la méthode de branchement local de Fischetti et Lodi [25]. La deuxième, est une méthode hybride utilisant deux stratégies complémentaires : la méthode de branchement local et la génération de colonnes. La troisième, est une version généralisée de la deuxième. Elle peut être considérée comme un algorithme de branch-and-bound tronqué dans lequel seulement certains nœuds élites sont traités par la méthode hybride.

III.3.1 Méthode de branchement local standard (BL)

La méthode de branchement local (séparation locale) est une méthode de résolution qui s'appuie principalement sur la recherche par voisinage. Elle est classée parmi les méthodes-méta-heuristiques bien qu'elle a été proposée à l'origine par Fischetti et Lodi [25] comme méthode exacte. Puis, elle a été employée comme heuristique à deux niveaux par Fischetti et al [27].

Une des particularités de cette méthode concerne la manière dont les voisinages sont constuits. En effet, ces derniers sont obtenus en introduisant des inégalités linéaires non-valides, appelées *coupes locales de branchement* ("local branching").

La linéarité des contraintes ajoutées offre alors la possibilité d'explorer les voisinages induits par des solveurs de programmes linéaires tels que Cplex. À ce jour, ces derniers ont prouvé leur efficacité à résoudre des programmes linéaires en nombres entiers et mixtes (de taille significative) en temps raisonable et ce de manière exacte et approchée. Toutefois, leur efficacité reste un peu limitée sur quelques problèmes complexes en ce qui concerne le temps de résolution ainsi que la qualité des solutions heuristiques. La méthode de branchement local vise, en réalité, à rendre ce type de solveur plus efficace en permettant d'obtenir de bonnes solutions au début de l'exploration de l'espace de recherche. Elle alterne deux types de branchements : un branchement à haut niveau pour définir les voisinages et un branchement à bas niveau pour les explorer en utilisant un solveur de programmes linéaires.

Principe de résolution par BL

La méthode de branchement local peut être décrite par les étapes suivantes :

1. Générer une solution initiale pour la première arborescence (nœud).
2. Initialiser la première arborescence en utilisant la solution précédemment générée.
3. Répéter les étapes :
   (a) Résoudre de façon complète l'arborescence locale
   (b) Lorsque la recherche locale est terminée, deux cas se présentent :
      i. Une meilleure solution réalisable a été obtenue dans cette arborescence locale. Alors créer une nouvelle arborescence utilisant la solution améliorée comme solution initiale (solution référence).
      ii. La solution n'a pas été améliorée, alors interrompre le branchement local.
4. Résoudre le reste de l'arbre de recherche.
5. Retourner la meilleure solution trouvée.

Dans [25], Fischetti et Lodi ont appliqué le branchement local au programme linéaire mixte en nombre entiers, de la forme générale suivants :

$$(MIP)\begin{cases} \text{Minimize} & c^T x & \\ & Ax \geq b & \\ & x_j \geq 0 & \forall j \in G,\ x_j \text{ entier} \\ & x_k \geq 0 & \forall k \in C \\ & x_i \in \{0,1\} & \forall i \in B \neq \emptyset, \end{cases}$$

où l'ensemble d'indices des variables $N := \{1, ..., n\}$ est divisé en trois sous-ensembles : B, G, C tels que :

- $B \neq \emptyset$ contient les indices des variables binaires.
- G et C, représentent les indices des variables entières et ceux des variables continues, respectivement.

Soient $\overline{x}$ une solution réalisable initiale, appelée *solution référence*, pour le problème MIP et k un paramètre entier positif.

Le K_{OPT} voisinage de $\overline{x}$ est l'ensemble des solutions réalisables du problème (MIP) satisfaisant additionnellement la contrainte dite *contrainte de branchement local* suivante :

$$\Delta(x,\bar{x}) \quad := \quad \sum_{j\in B \ | \ \bar{x_j}=1} (1-x_j) + \sum_{j\in B \ | \ \bar{x_j}=0} x_j \quad \leq \quad k \tag{III.5}$$

Les deux termes gauches de l'inégalité (III.5) représentent le nombre de variables binaires de la solution $\overline{x}$ qui passent de 1 à 0 ou de 0 à 1, respectivement. Notons que cette contrainte représente aussi la distance de Hamming maximale d'ordre k parmi les solutions voisines de x.

Lorsque la cardinalité de l'ensemble β est connue d'avance, l'inégalité (III.5) s'ecrit de la façon suivante :

$$\Delta(x,\bar{x}) := \sum_{j\in B \ | \ \bar{x_j}=1} (1-x_j) \leq k' \left(= \frac{k}{2}\right) \tag{III.6}$$

La contrainte de branchement local peut être utilisée comme un critère de branchement dans un schéma énumératif pour (MIP) (voir Fischetti et Lodi [25]). En effet, étant donnée une solution référence $\overline{x}$, le sous espace correspondant au nœud en cours de traitement peut être divisé suivant les deux contraintes disjointes suivantes :

$$\Delta(x,\bar{x}) \leq k \quad \text{(branche gauche)} \qquad \text{ou} \qquad \Delta(x,\bar{x}) \geq k+1 \quad \text{(branche droite)},$$

où la contrainte droite est appelée *branche normale*.

Un solveur de programmes linéaires est considéré ensuite comme boite noire pour résoudre le voisinage induit par la branche gauche (branche locale). De ce fait, le choix du paramètre k, représentant la taille du voisinage, doit être adéquat. Sa valeur doit être d'un côté, assez petite pour que le voisinage soit exploré en temps raisonable. D'un autre côté, elle doit être assez grande pour assurer que le voisinage contient des solutions meilleures que $\bar{x}$.

Lorsque la branche gauche est complétement résolue, la solution obtenue $\bar{x}^1$ devient la nouvelle solution référence. Par la suite, l'espace de recherche correspondant à la branche normale est séparé et les deux nouvelles branches suivantes sont obtenues :

$$\Delta(x,\bar{x}) \geq k+1,\ \Delta(x,\bar{x}^1) \leq k \quad \text{(branche locale)},$$

et

$$\Delta(x,\bar{x}) \geq k+1,\ \Delta(x,\bar{x}^1) > k \quad \text{(branche normale)},$$

Le processus de recherche continue en alternant des branches locales et normales jusqu'à ce que la solution référence ne puisse plus être améliorée.

D'autres extensions de l'algorithme de branchement local ont été proposées par Fischetti et Lodi [25] comme, par exemple, limiter le temps de résolution des branches locales ou utiliser des techniques de diversification lorsque celles-ci n'atteignent pas de meilleures solutions.

En ce qui concerne le MMKP, toute solution x a exactement n variables fixées à 1. Ainsi, pour une solution de référence $\bar{x}$, la contrainte locale s'écrit comme suit :

$$\Delta(x, \bar{x}) \quad := \quad \sum_{j \in S \ | \ \bar{x_j}=1} (1 - x_j) \quad \leq \quad \frac{k}{2} \tag{III.7}$$

Dans notre adaptation de l'algorithme de branchement local pour la résolution de MMKP, nous employons le solveur Cplex pour résoudre les voisinages (programmes linéaires) développés. Nous limitons le nombre de nœuds lors de la résolution par Cplex et nous fixons également un temps-limite pour l'algorithme de branchement local. Pour plus de détails, nous renvoyons à la section III.3.4 où les résultats de BL, sont reportés dans le tableau III.7 (voir aussi Cherfi et Hifi [11], [12]).

III.3.2 Un algorithme de branchement local hybride pour le MMKP (BLH)

Nous allons présenter, dans cette partie, un algorithme hybride pour résoudre de façon approchée le MMKP. Ce dernier repose sur la méthode de branchement local et la méthode de génération de colonnes. Il s'agit de remplacer la méthode d'énumération implicite du branchement local standard par : (i) la méthode de génération de colonnes pour résoudre les programmes linéaires relâchés et (ii) la procédure d'arrondi hybride PAH décrite dans la section III.2.3.2 pour réduire l'espace de recherche. Ces procédures sont hybridées afin d'accélérer le processus de recherche et d'améliorer la qualité des solutions obtenues. Comme le branchement local exploite des branchements sur contraintes en créant une sorte d'intensification et de diversification, nous l'utilisons à chaque fois que la solution référence est améliorée lors de la résolution par la méthode PAH.

III.3.2.1 Calcul d'une solution initiale

La solution initiale que nous considérons est obtenue en appliquant l'algorithme de recherche locale réactive (RLS), proposé par Hifi et al [39]. Nous utilisons cette solution afin de :

- construire une solution de départ pour le branchement local.
- Initialiser le processus de génération de colonnes avec les colonnes de cette solution (et d'autes colonnes).

La méthode RLS est caractérisée par la période d'interdiction détérminée à travers le mécanisme de rétroaction durant la recherche. Ce principe tente de débloquer une solution qui cycle. L'algorithme simule ce procédé en considérant la génération de solutions à deux phases. La première phase consiste à dégrader la solution courante qui n'est pas "assez" améliorée, et applique des interchanges locaux entre les éléments de la solution dégradée dans le but d'atteindre une meilleure solution locale. La deuxième phase consiste à débloquer la solution pour diversifier la recherche en changeant de direction pour l'exploration d'autres régions. Nous notons la solution obtenue par F.

Pour initialiser la génération de colonnes, nous utilisons les colonnes de F et nous rajoutons aussi une colonne par classe i, $i = 1, \ldots, n$ (différentes de celles de F) correspondant au plus grand rapport pseudo-utilité :

$$\max_{1 \leq j \leq r_i} \frac{v_{ij}}{\sum_{k=1}^{m} w_{ij}^k}.$$

Ceci veut dire que nous favorisons la colonne réalisant la plus grande utilité relativement à toutes les contraintes knapsack.

III.3.2.2 Description de l'algorithme BLH

À l'instar de Fischetti et Lodi [25], nous présentons l'idée principale de l'algorithme BLH dans un exemple générique. Dans ce contexte, la figure III.4 résume le mécanisme de branchement local.

Nous commençons par calculer une solution réalisable $\overline{x}$ par la méthode décrite dans III.3.2.1. Cette dernière sera associée au nœud racine (nœud1 de la figure III.4).

Chaque nœud développé crée à son tour deux successeurs : un nœud représentant la branche gauche de l'arborescence et un nœud droit représentant la branche droite.

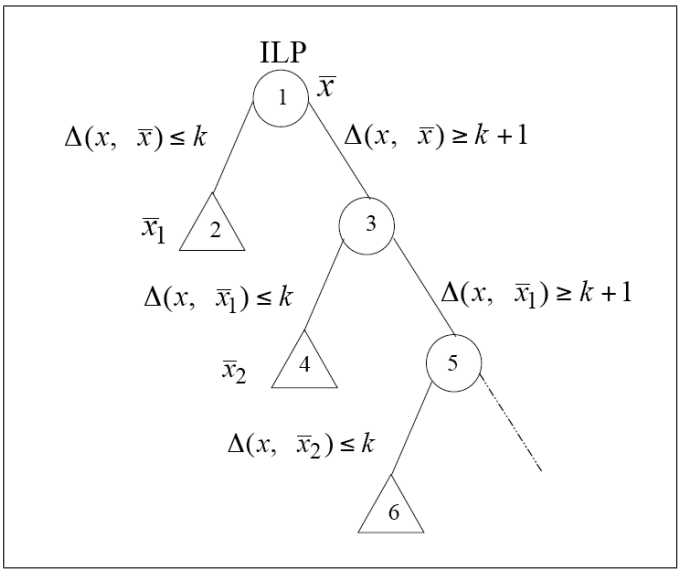

FIG. III.4 : Schéma du branchement local hybride : BLH

La branche gauche correspondant au rajout de la contrainte locale :

$$\Delta(x, \bar{x}) \leq k,$$

au programme linéaire restreint en nombres entiers, notée ILP, associé au nœud en cours de traitement. Le voisinage induit par cette dernière est ensuite résolu par la méthode PAH. Ainsi, nous distinguons les deux cas suivants après la résolution :

(1) Une solution $\overline{x}_1$ (meilleure que $\bar{x}$) émerge de la branche gauche.

(2) PAH ne parvient pas à améliorer la solution $\overline{x}$.

• Considérons tout d'abord le cas (1) ; la solution $\overline{x}_1$, obtenue dans le K_{OPT} voisinage $N(\overline{x}, k)$ associé à $\overline{x}$, est une amélioration de $\overline{x}$. Désormais, elle devient la nouvelle solution référence et par la suite le même processus est appliqué à la branche droite (nœud 3, figure III.4). Nous supposons que PAH atteint une meilleure solution $\overline{x}_2$, lors de l'exploration du voisinage $N(\overline{x}_1, k) \setminus N(\overline{x}, k)$ (nœud 4, figure III.4). Alors le nœud 5 associé au problème original ILP, augmenté par les deux contraintes suivantes :

$$\Delta(x, \overline{x}) \geq k+1 \quad \text{and} \quad \Delta(x, \overline{x}_1) \geq k+1.$$

subira le même traitement.

• Considérons maintenant le cas (2) où la méthode PAH n'améliore pas la solution référence. Supposons, par exemple, qu'on se trouve face à cette situation au dernier nœud.

Alors, suivant notre description dans la figure III.4, au nœud 5 la solution $\overline{x}_2$ n'a pas été améliorée. Nous procédons dans ce cas selon le résultat procuré par la procédure PAH. Nous distinguons deux cas :

1. Il n' y a pas eu d'amélioration : Nous tentons d'enrichir l'espace de recherche en appliquant une stratégie d'intensification. Dans ce cas, la stratégie employée consiste à réitérer la première phase de la méthode PAH en fixant un sous-ensemble différent de variables à 1.
2. Le processus ne parvient pas à determiner une solution réalisable : Nous tentons une diversification afin d'explorer d'autres segments de l'espace de recherche.

Les étapes principales de l'algorithme de branchement local hybride BLH sont présentées dans la figure III.5. Quant aux stratégies d'intensification et de diversification employées, elles feront l'objet des paragraphes qui suivent.

III.3.2.3 Techniques de diversification et intensification

Nous décrivons dans cettte section les méthodes d'intensification et de diversification employées dans l'algorithme BLH. Ces dernières deviennent nécessaires lorsque la méthode PAH ne parvient pas à améliorer la solution référence en cours.
Soit $\overline{S}$ l'ensemble des variables fixées à 1 après une résolution par la méthode PAH. On considère la partition $(\overline{B}_1, \overline{B}_1)$ tel que $\overline{S}_1 = \overline{B}_1 \cup \overline{B}_1$ et $\overline{B}_1 \cap \overline{B}_2 = \emptyset$ où :

- $\overline{B}_1$ contient les indices des variables fixées à 1 après la première phase (la phase d'arrondi) de PAH.
- $\overline{B}_2$ contient les indices des variables fixées à 1 lorsque la deuxième phase de PAH (l'algorithme exact tronqué) est appliquée.

a) Technique d'intensification

Étant donnée une solution $\overline{x}$ associée à un problème restreint ILP, nous tentons d'explorer efficacement son voisinage en fixant au plus toutes les variables indexées dans $\overline{B}_1$ à 1. Pour ce faire, nous introduisons une stratégie d'intensification composée de deux phases complémentaires :

La première phase :

Dans cette phase, nous injectons au problème ILP la contrainte de branchement local

Entrée : `Une instance du MMKP.`

Sortie : `Une solution approchée pour le MMKP.`

(I) **Étape d'initialisation.**

Générer une solution réalisable pour le problème restreint ILP et initialiser la première arbrescence locale par cette solution.

(II) **Étape itérative.**

Répéter

(a) Ajouter une (des) branche(s) à l'arborescence correspondante au problème restreint ILP.

(b) Faire appel à la procédure PAH pour résoudre le problème ILP.

(c) Évaluation de la solution obtenue (après la recherche locale) :

i. *Une solution meilleure à été atteinte* :
Créer une nouvelle arborescence locale en utilisant comme solution de départ, la solution obtenue par l'arborescence locale en cours.

ii. *Aucune solution meilleure n'a été obtenue* :
Appliquer les méthodes d'intensification et de diversification pour enrichir et/ou restreindre l'espace de recherche.

Jusqu'à satisfaction de la condition d'arrêt.

(III) **Étape finale.**

Retourner la meilleure solution trouvée.

Fig. III.5 : Étapes générales de l'algorithme BLH

suivante :

$$\Delta_1(x, \bar{x}) := \sum_{j \in \bar{B}_1} (1 - x_j) \leq k_1.$$

Par la suite, nous résolvons le problème ainsi obtenu avec la méthode PAH. Selon la nouvelle solution calculée, deux cas existent :

- Dans le premier, la nouvelle solution est meilleure que la solution $\overline{x}$. Alors, cette dernière deviendra la nouvelle solution référence et la contrainte Δ_1 sera supprimée de ILP.
- Dans le deuxième cas, le processus de recherche n'est pas parvenu à atteindre une solution réalisable ou bien, le problème est devenu non réalisable à cause de la contrainte Δ_1. Donc, nous appliquons la deuxième phase décrite ci-dessous. Notons ici que la contrainte Δ_1 doit être supprimée de ILP si ce dernier devient non faisable.

La deuxième phase :

Dans cette deuxième phase, nous essayons de perturber les éléments déjà fixés à 1 par la méthode PAH. Tout d'abord, nous rappelons que la contrainte Δ_1 représente toujours une contrainte du modèle courant. Nous limitons ensuite le nombre des variables de décision fixée à 'un' dans la seconde phase de PAH (variables indexées dans $\overline{B}_2$).

Nous procédons en suivant les étapes ci-après :

1. Considérer les paramètres suivants :
   - k_2 : le nombre de variables indexées dans $\overline{B}_2$ et forcées à 1. Ce paramètre est initialisé à 0 ;
   - k_{step} : paramètre-étape utilisé pour élargir l'espace de recherche ;
   - k_{max} : taille limite du voisinage (valeur maximale de k_2) ;
2. *Répéter les étapes* :
   (a) Injecter la contrainte
   $$\Omega := \sum_{j \in \bar{B}_2} x_j = k_2$$
   et résoudre le problème résultant par la méthode PAH.
   (b) Soit $\hat{x}$ la solution fournie par PAH. Alors, deux cas existent :
      i. $\hat{x}$ est une amélioration de $\bar{x}$. Dans ce cas, mettre à jour la solution référence en remplaçant $\bar{x}$ par $\hat{x}$ et réinitialiser le paramètre k_2 à 0.
      ii. $\bar{x} = \hat{x}$. Alors, supprimer la contrainte Ω et élargir la taille du voisinage en posant $k_2 = k_2 + k_{step}$.

   jusqu'à $(k_2 = k_{max})$.
3. Sortir avec la meilleure solution $\bar{x}$.

Il est important de noter ici qu'après la méthode d'intensification, les contraintes Δ et Ω seront supprimées du modèle en cours.

b) Technique de diversification

On emploie une stratégie de diversification lorsque le processus de recherche (dans la branche gauche) ne parvient pas à améliorer la solution de référence. En effet, la diversification dans ce cas est nécéssaire pour explorer d'autres espaces de recherche ou simplement compléter l'espace de recherche en cours. Ici, la stratégie utilisée consiste à élargir la taille du voisinage en augmentant le nombre de variables de décisions, pouvant

être fixées à 1, par au plus $\lceil k/2 \rceil$. Par conséquent, une nouvelle branche-gauche est créée et la contrainte suivante est injectée

$$\Delta(x, \overline{x}) \leq k + \left\lceil \frac{k}{2} \right\rceil.$$

La procédure PAH sera ensuite appliquée au problème qui en résulte (contenu dans ce nœud). S' il n' y pas eu d'amélioration, nous pouvons imposer une contrainte tabou. Nous appliquons les étapes suivantes :

1. Rajouter la contrainte tabou suivante au problème :

$$\Delta_1(x, \bar{x}) \geq 1.$$

   Ceci à pour but de modifier la structure de la solution courante.

2. Résoudre le problème résultant en appliquant PAH et considérer la solution obtenue comme la nouvelle solution de référence.

III.3.3 Branchement local hybride généralisé pour le MMKP (BLHG)

Nous allons présenter dans cette partie une adaptation de la méthode décrite dans la section III.2.3.3. C'est une généralisation du branchement local hybride (voir section III.3.2). Elle consiste à appliquer un algorithme de branch-and-bound restreint dans lequel seulement quelques nœuds (appelés nœuds élites) seront résolus par la procédure BLH. Nous utilisons cette technique pour résoudre le MMKP dans un algorithme à trois niveaux décrit dans la figure III.6.

Notons que l'algorithme BLHG simule un algorithme exact sans garantir l'optimalité de la solution qu'il fournit. Par ailleurs, ce dernier nécessite une solution initiale et des nœuds élites pour lesquels la procédure BLH sera appliquée.
La méthode de Hifi et al [39] est utilisée pour calculer la solution initiale. Elle permet également de construire le premier problème restreint ILP (voir la section III.3.2.1). Ce dernier correspond au nœud racine de l'arbre de branch-and-bound.
D'autres nœuds sont ensuite créés en alternant la première et la deuxième séparation présentées dans la section III.2.3.4. Pareillement à la procédure d'arrondi hybride augmentée (section III.2.3.3), les noeuds élites dans l'étape 2a sont choisis en introduisant deux paramètres empiriques β_1 et β_2. Le paramètre β_1 représente le nombre de nœuds développés

Entrée : `Une instance du MMKP.`

Sortie : `Une solution approchée pour le MMKP.`

1. Construire une solution réalisable *initiale* du problème original MMKP.
2. **Tant que** (test-d'arrêt-non-vérifié) **faire**
   (a) Séléctionner le *nœud actif* η (représentant le nœud racine au début) correspondant au *problème restreint* ILP associé au MMKP.
   (b) Appliquer la procédure BLH (décrite dans la Section III.3.2) au problème courant restreint ILP.
   (c) Créer de nouveaux nœuds en utilisant quelques *stratégies de branchement* et supprimer les nœuds dont la valeur objectif est inférieure ou égale à la valeur de la meilleure solution réalisable.

   Fin Tant que
3. Retourner la meilleure solution.

Fig. III.6 : Étapes générales de l'algorithme BLHG

et β_2 la fréquence avec laquelle BLH sera appliquée.
Plus de détails sont reportés dans la partie expérimentale qui fera l'objet de la section suivante.

III.3.4 Résultats expérimentaux

Dans cette partie, nous présentons une étude expérimentale des algorithmes proposés. Notre objectif, d'un côté, est d'évaluer les performances de ces derniers. D'un autre côté, nous montrons l'impact qu'a la génération de colonnes sur la qualité des solutions. Dans ce contexte, nous effectuons une étude comparative des résultats fournis par ces algorithmes avec les meilleurs résultats connus dans la littérature.
Nous rappelons que trois versions de branchement local ont été présentées : BL, BLH et BLHG.

- BL : algorithme de branchement local standard adapté au MMKP.
- BLH : algorithme hybridant LB avec la méthode PAH (qui est une combinaison entre une procédure d'arrondi et une génération de colonnes)
- BLHG : généralisasion de BLH (BLH appliqué à quelques nœuds séléctionnés d'un

branch-and-bound tronqué)

Tous ces algorithmes sont codés en langage `C++` et testés sur une station `Ultra-Sparc10` (250Mhz et avec 128Mb de RAM).

Les tests sont effectués sur les instances utilisées par Hifi et al [39] et celles utilisées par Khan et al [47]. Nous testons un total de 33 instances divisées en deux groupes différents. Le premier correspond aux instances de Khan et al [47]. Il contient 13 instances (notées I01, ..., I13). Le deuxième groupe est composé de 20 instances, notées Ins1, ..., Ins20, utilisées dans Hifi et al [39], et variant de taille moyenne à taille large.

III.3.4.1 Effet de la génération de colonnes sur le branchement local

Dans cette partie, nous comparons les performances des procédures BL et BLH sur 33 instances de la littérature. Pour démontrer l'effet de la génération de colonnes, nous fixons le même temps-limite de résolution pour BL et BLH. Puis, nous analysons le comportement de chaque algorithme.

Notons que l'emploi des méthodes approchées pour la résolution des problèmes d'optimisation combinatoire entraîne l'utilisation de différents paramètres. De plus, les valeurs attribuées à ces derniers influencent visiblement la qualité des solutions obtenues.

Les algorithmes BL et BLH nécessitent plusieurs paramètres décrits avec les valeurs qui leurs sont attribuées ci-dessous :

1. La taille des voisinages construits. Nous la fixons en posant $k = \lceil \frac{n}{4} \rceil$.
2. Le nombre de nœuds à explorer lorsque le solveur Cplex est utilisé pour résoudre les voisinages est fixé comme suit :
   - Pour BL, nous considérons un maximum de 5000 nœuds par arborescence.
   - Pour BLH, nous considérons 3000 nœuds par arborescence.
3. Concernant BLH, nous devons fixer les paramètres utilisés dans l'intensification et dans la méthode PAH. Nous procédons comme suit :
   - Pour l'intensification, nous considérons les valeurs suivantes :
     $k_1 = |\bar{B}_1| - 2$, $k_{max} = 10$ et $k_{step} = 2$.
   - Comme décrit dans la séction III.2.3.2 et dans Cherfi et Hifi [12], PAH nécessite de définir le pourcentage des variables à fixer à 1 dans les ensembles I_1 et I_2. L'ensemble I_1 contient les classes fixées (c'est-à-dire les classes ayant un élément

fixé à 1) et l'ensemble I_2 contient les classes libres.

Nous utilisons deux paramètres, α_1 et α_2 où α_1 représente le pourcentage maximum de classes à fixer à 1 et α_2 le pourcentage de classes à fixer dans I_1.

Soit $\alpha = \alpha_1 - \alpha_2$ un paramètre représentant le pourcentage des variables fractionaires à fixer à 1 dans I_2. Alors, nous suivons le même schéma décrit dans la section III.2 en fixant les deux paramètres α_1 et α_2 à 50% (voir aussi Cherfi et Hifi([11], [12])).

Nous exécutons les deux algorithmes en limitant le temps de résolution à $t_1 = 300\,s$ puis, $t_2 = 600\,s$ et enfin $t_3 = 1200\,s$. Ensuite, nous comparons, pour chaque temps-limite imposé, la qualité des solutions obtenues. Ces dernières sont reportées dans le tableau III.7 où les noms des instances traitées sont donnés dans la colonne 1. Les meilleurs résultats de la littérature extraits de Cherfi et Hifi([11], [12]) sont reportés dans la colonne 2. La colonne 3 contient les solutions initiales utilisées. Les résultats de BL sont reportés dans les colonnes 4, 5 et 6 correspondant, respectivement, au temps-limite $t_1 = 300\,s$, $t_2 = 600\,s$ et $t_3 = 1200\,s$. Les résultats de BLH sont reportés dans les colonnes 7, 8 et 9 correspondant à $t_1 = 300\,s$, $t_2 = 600\,s$ et $t_3 = 1200\,s$, respectivement. Dans le tableau III.7, le symbole $\circ$ signifie que la solution optimale n'est pas connue et $\star$ signifie que l'algorithme fournit une meilleure solution.

Du tableau III.7, nous observons les points suivants :

1. *L'analyse de BL* : nous comparons la qualité des solutions obtenues aux solutions initiales. BL parvient à améliorer 8 instances parmi 27 pour $t_1 = 300$ (colonne 4). Notons ici que les six premières instances sont résolues jusqu'à l'optimalité. Lorsque le temps-limite $t_2 = 600$ (colonne 5), BL améliore 14 autres instances. Finalement, l'amélioration devient plus importante lorsque le temps-limite $t_3 = 1200$ (colonne 6). En effet, dans ce cas, l'amélioration concerne 13 autres solutions.
2. *L'analyse de BLH* : Pareillement à BL, en accroissant le temps-limite fixé, le nombre de solutions améliorées par BLH augmente. En effet, BLH améliore 9 instances parmi 27 lorsque le temps-limite est fixé à 300 s. Le nombre de solutions améliorées augmente à 22 lorsque le temps-limite est fixé à 600 s et devient encore plus important pour $t_3 = 1200\,s$ (24 améliorations).

Notons que, d'un côté, BL parvient à atteindre deux nouvelles meilleures solutions (la

			BL			BLH		
#Inst.	Opt/Meil	Sol.init	t_1	t_2	t_3	t_1	t_2	t_3
Group 1.								
I01	173	173	–	–	–	–	–	–
I02	364	364	–	–	–	–	–	–
I03	1602	1602	–	–	–	–	–	–
I04	3597	3597	–	–	–	–	–	–
I05	3905.70	3905.70	–	–	–	–	–	–
I06	4799.30	4799.30	–	–	–	–	–	–
I07°	24587	24587	24587	24587	24587	24587	24587	24587
I08°	36892	36877	36877	36877	36878	36877	36877	36878
I09°	49176	49167	49167	49171	49171	49167	49171	49171
I10°	61461	61437	61440	61445	61449	61440	61449	61450⋆
I11°	73775	73773	73773	73775	73775	73773	73775	73777⋆
I12°	86078	86069	86070	86070	86078	86070	86078	86078
I13°	98431	98429	98429	98429	98431	98429	98431	98431
Group 2.								
Ins01°	10732	17014	10714	10714	10719	10714	10719	10732⋆
Ins02°	13598	13598	13598	13598	13598	13598	13598	13598
Ins03°	10943	10943	10943	10943	10943	10943	10944⋆	10944⋆
Ins04°	14440	14429	14430	14430	14432	14430	14430	14432
Ins05°	17053	17053	17053	17053	17053	17053	17053	17053
Ins06°	16825	16823	16823	16825	16826	16825	16825	16826⋆
Ins07°	16435	16423	16427	16429	16432	16427	16429	16432
Ins08°	17510	17506	17506	17507	17507	17506	17508	17508
Ins09°	17760	17754	17756	17756	17756	17756	17758	17758
Ins10°	19314	19314	19314	19315⋆	19315⋆	19314	19315⋆	19315⋆
Ins11°	19434	19431	19431	19431	19434	19431	19431	19434
Ins12°	21731	21730	21730	21731	21731	21730	21731	21731
Ins13°	21575	21569	21570	21571	21571	21571	21571	21571
Ins14°	32870	32869	32870	32870	32870	32869	32870	32871⋆
Ins15°	39157	39148	39148	39149	39151	39148	39149	39151
Ins16°	43361	43354	43356	43357	43357	43354	43357	43357
Ins17°	54349	54349	54349	53450⋆	53454⋆	54350⋆	54350⋆	53454⋆
Ins18°	60460	60456	60456	60457	60457	60456	60457	60457
Ins19°	64923	64921	64921	64921	64923	64921	64924⋆	64924⋆
Ins20°	75611	75603	75603	75604	75608	75604	75608	75609⋆

Tab. III.7 : Effet de la génération de colonnes : performances de BLH et de BL

valeur de la solution est marquée dans ce cas par le symbole $*$: Ins10 et Ins17). D'un autre côté, la méthode BLH, spécialement pour t_3, parvient à atteindre 10 nouvelles meilleures solutions parmi 27. Nous pouvons conclure que, pour les deux groupes d'instances, BLH domine le branchement local standard BL.

III.3.4.2 Performances de l'algorithme de BLHG

Dans cette partie, nous comparons les résultats fournis par l'algorithme généralisé BLHG aux :

- Résultats du solveur Cplex après une heure de résolution.
- Les meilleures solutions produites par l'algorithme proposé par Cherfi et Hifi([12], [11]), noté Algo.
- Les meilleures solutions du tableau III.7 (marquées par le symbole $*$, colonne 9).

Nous utilisons les mêmes paramètres de l'algorithme BLH à l'exception du nombre de nœuds utilisés pour explorer les voisinages par Cplex. Nous le fixons à 1000 nœuds par arborescence et nous limitons le nombre d'intensification et de diversification utilisées à 10. Nous considérons également un temps-limite de 100 secondes pour chaque branchement local. Les valeurs de β_1 et β_2, utilisées dans Cherfi et Hifi([12], [11]) et décrites dans la section III.2.4 sont retenues. Ainsi, le nombre de nœuds de l'algorithme généralisé est fixé à $\beta_1 = 500$ et le paramètre de fréquence $\beta_2 = 10$. Notons que ce dernier représente le nombre de séparations effectuées avant de développer le dernier nœud élite. Notons aussi que générer β_1 nœuds signifie que $\beta_1/2$ nœuds peuvent être traités par BLH vu qu'à chaque séparation, deux nœuds sont créés.
Nous excécutons l'algorithme BLHG en considérant deux temps-limites d'exécution $t_1 = 600\,s$ et $t_2 = 1200\,s$. Pour chaque temps de résolution, nous considérons également un temps-limite (de 30 secondes) pour la technique d'intensification - si cette dernière est utilisée -.

L'algorithme BLHG est évalué sur deux groupes d'instances et les résultats sont reportés dans le tableau III.8. Dans la colonne 2 sont représentées les meilleures solutions atteintes par Algo. Dans le cas où ces solutions sont optimales, les instances correspondantes seront marquées par le symbol '$\diamond$'. La colonne 3 contient les solutions du solveur Cplex. La colonne 4 contient les meilleures solutions de l'algorithme BLH (colonne 9 du tableau III.7). Les colonnes 6 et 7 représentent les solutions fournies par l'algorithme BLHG pour le temps-limite t_1 et t_2, respectivement. Dans le tableau III.8, le symbole "$-$" signifie que l'algorithme atteint la solution optimale.

L'étude du tableau III.8, montre que :

- Pour les deux temps de résolution t_1 et t_2, BLHG fournit de meilleures solutions par

				BLHG	
#Inst.	Algo	Cplex	Meil (Tab. III.7)	t_1	t_2
Group 1.					
I01$^\diamond$	173	–	–	–	–
I02$^\diamond$	364	–	–	–	–
I03$^\diamond$	1602	–	–	–	–
I04$^\diamond$	3597	–	–	–	–
I05$^\diamond$	3905.70	–	–	–	–
I06$^\diamond$	4799.30	–	–	–	–
I07	24587	24584	24587	24587	24587
I08	36892	36869	36878	36894*	36894*
I09	49176	49155	49171	49179*	49179*
I10	61461	61446	61450	61464*	61464*
I11	73775	73759	*73777*	*73777*	73783*
I12	86078	86071	86078	86080*	86080*
I13	98431	98418	98431	*98433*	98438*
Group 2.					
Ins01	10732	10709	10732	10738*	10738*
Ins02	13598	13597	13598	13598	13598
Ins03	10943	10934	*10944*	*10944*	*10944*
Ins04	14440	19422	14432	14442*	14442*
Ins05	17053	17041	17053	17053	17053
Ins06	16825	16815	*16826*	16827*	16827*
Ins07	16435	16407	16432	16440*	16440*
Ins08	17510	17484	17508	17510	17510
Ins09	17760	17747	17758	17761*	17761*
Ins10	19314	19285	*19315*	19316*	19316*
Ins11	19434	19424	19434	19441*	19441*
Ins12	21731	21725	21731	21732*	21732*
Ins13	21575	21569	21571	21577*	21577*
Ins14	32870	32866	*32871*	32874*	32874*
Ins15	39157	39154	39151	39157	39160*
Ins16	43361	43357	43357	43362*	43362*
Ins17	54349	54349	*54354*	*54352*	54360*
Ins18	60460	60455	60457	60460	60464*
Ins19	64923	64919	*64924*	64925*	64925*
Ins20	75611	75603	75609	75612*	75612*

Tab. III.8 : Performance de l'algorithme de branchement local hybride généralisé BLHG

rapport au solveur Cplex et domine les solutions atteintes par BLH.

- Pour t_1, l'algorithme BLHG produit 17 meilleures solutions parmi 27 comparées à celles atteintes par le solveur Cplex et BLH. Notons que pour les six premières instances, tous les algorithmes produisent la solution optimale.
- Pour t_2, le nombre de solutions améliorées augmente. En effet, cette version parvient

à améliorer cinq autres solutions et donc, le nombre de solutions améliorées devient plus important (22 sur 27 instances).

III.4 Conclusion

Dans ce chapitre, nous avons présenté principalement deux méthodes de résolution approchée pour le MMKP. Pour chaque méthode, nous avons présenté plusieurs algorithmes. La première méthode est basée principalement sur la génération de colonnes et une technique d'arrondi. Nous avons commencé par présenter une procédure d'arrondi gloutonne. Nous avons par la suite présenté un autre algorithme hybride composé de deux procédures complémentaires : (i) une procédure d'arrondi pour fixer une partie des variables et (ii) un algorithme exact restreint qui résout le problème réduit. Après expérimentation, nous avons constaté que la première procédure d'arrondi est capable de produire de bonnes solutions en un temps de calcul très modeste. La procédure d'arrondi hybride permet d'obtenir de meilleures solutions et ce en temps raisonable.
La deuxième méthode approchée est basée sur le branchement local. Nous avons présenté trois algorithmes. Le premier est une adaptation du branchement local standard au MMKP. Le deuxième est une hybridation de la génération de colonnes avec le branchement local standard pour résoudre efficacement les instances de taille large. Enfin, le troisième généralise le deuxième algorithme en l'introduisant dans un algorithme de branch-and-bound tronqué où il est appliqué à un sous-ensemble de nœuds sélectionnés. Les performances de ces trois méthodes ont été testées sur des instances de la littérature et comparées aux meilleurs résultats connus. La première version reste compétitive comparée aux résultats de la littérature. La deuxième version tire profit de la génération de colonnes et arrive à améliorer la plupart des instances. La dernière domine les deux autres versions vu la qualité des solutions qu'elle fournit mais nécessite plus de temps de calcul (qui reste raisonnable).

Chapitre IV

Résolution exacte du problème du knapsack généralisé à choix multiple

Dans ce chapitre nous nous intéressons à la résolution exacte du problème de sac-à-dos généralisé à choix multiple. Nous proposons une méthode de résolution exacte en se basant sur une procédure de "branch-and-cut". Le principe de la méthode repose sur (i) la détermination de nouvelles contraintes valides et (ii) l'utilisation de contraintes de couverture locales et globales. Ces dernières sont construites à partir de couvertures minimales dont les des éléments appartiennent à des classes différentes.

IV.1 Introduction

En général, un algorithme exact devient efficace lorsqu'on lui associe des bornes inférieures et supérieures de bonne qualité. Jusqu'à présent, et en particulier pour le problème de sac-à-dos généralisé à choix multiple, les bornes utilisées s'appuyaient principalement sur la méthode de la relaxation lagrangienne (Sbihi [72]) ou de la relaxation "surrogate" (Hernandez [37]). Cependant, aucune étude n'a été effectuée par adaptation des méthodes des plans coupants. Le but de ces méthodes est de tenter la génération de bornes plus fines par réduction de l'espace de recherche et d'accélérer la résolution du problème traité.

Dans ce chappitre, nous nous intéressons à la résolution exacte du problème de sac-à-dos généralisé à choix multiple par application d'une procédure de coupe.
Dans un premier temps, nous commençons par proposer des contraintes valides pour

le problème traité. Dans un deuxième temps, nous proposons d'utiliser des contraintes de couverture étendues globales qui prennent en considération toutes les contraintes du problème. Par la suite, nous décrivons un algorithme de "branch-and-cut" en introduisant les contraintes précédentes dans un schéma énumératif.
Finalement, nous présentons une étude expérimentale pour valider la méthode proposée sur deux groupes d'instances : un premier groupe composé d'instances de la littérature (extraites de Khan et al. [47]) et un deuxième groupe composé d'autres instances générées aléatoirement.

IV.2 Inégalités de couverture étendues : locales et globales

Dans cette section, nous proposons des inégalités de couverture étendues pour le MMKP. Ces dernières ont été introduites pour la première fois par Balas [2] et Wolsey [82]. Ensuite, elles ont été étudiées et généralisées par d'autres auteurs tels que Atamtürk [1] et Gu et al [36]. Récemment, d'autres contraintes de couverture étendues, dites *contraintes de couverture globales*, ont été proposées par Kontantinos et Letchford [50]. Ces derniéres ont été appliquées pour la résolution du problème de sac-à-dos multidimensionnel (MDKP : *Multi-Dimensional Knapsack Problem*) en variables binaires. Elles diffèrent des contraintes de couverture ordinaires (locales) par le fait de considérer toute la matrice associée aux contraintes.

Dans un premier temps, nous commençons cette section par l'introduction de quelques définitions et notations qui vont nous servir par la suite. Dans un deuxième temps, nous présentons les contraintes de couverture étendues (locales et globales) de la littérature. Ensuite, nous discutons l'adaptation de ces contraintes pour le MMKP ainsi que la façon de les générer. Enfin, nous présentons la technique de séparation employée pour les utiliser comme des plans coupants.

IV.2.1 Notations et définitions

Afin d'illustrer les inégalités de couverture, nous considérons le problème linéaire multi-contraint, que l'on notera LP, définie comme suit :

$$(LP)\begin{cases} \max\ z = & c.x \\ \text{s.c.} & Ax \leq b \\ & x \in \{0,1\}^n \end{cases}$$

où $c \in \mathbb{Z}_+^n$, $A \in \mathbb{Z}_+^{m\times n}$ est la matrice des coefficients des contraintes. Nous utilisons les notations suivantes :

- L'ensemble des solutions réalisables de la relaxation du problème (LP) est noté par

$$P := \{x \in [0,1]^n \ : \ Ax \leq b\}.$$

- L'enveloppe convexe des solutions réalisables entières de (LP) est représentée par

$$P_I := \text{conv}\{x \in \{0,1\}^n \ : \ Ax \leq b\}.$$

- Nous supposons que la $i^{ème}$ contrainte de (LP) (c-à-d la $i^{ème}$ contrainte knapsack) a la forme suivante :

$$\sum_{j=1}^{n} a_{ij}x_j \leq b_i. \tag{IV.1}$$

- *À* la contrainte (IV.1) est associé le polytope du sac-à-dos, noté Q_i, où

$$Q_i := \text{conv}\{x \in \{0,1\}^n \ : \ \sum_{j=1}^{n} a_{ij}x_j \leq b_i\}.$$

Alors, nous avons

$$P_I \subseteq \bigcap_{i=1}^{m} Q_i \subseteq P.$$

Définition IV.1. *Considérons le polytope Q_i, et soit $C \subseteq N = \{1,\ldots,n\}$. On dit que C est une couverture, si $\sum_{j\in C} a_j > b_i$. Pour toute couverture C, l'inégalité*

$$\sum_{j\in C} x_j \leq |C| - 1 \tag{IV.2}$$

est valide pour Q_i.

Définition IV.2. *Soit D un ensemble de N. Une contrainte du type $\sum_{j\in D} x_j \leq 1$ est appelée GUB-contrainte.*

Définition IV.3. *Considérons le polytope Q'_i et un ensemble K de GUB-contraintes disjointes tels que :*

$$Q'_i := conv\Big\{x \in \{0,1\}^n : \sum_{j=1}^{n} a_{ij}x_j \leq b_i, \sum_{j\in D_k} x_j \leq 1 \; \forall k \in K, D_k \cap D'_k = \emptyset \; \forall k \neq k', \bigcup_{k\in K} D_k = N\Big\}.$$

Un ensemble $C \subseteq N$ est appelé GUB-couverture si c'est une couverture pour la contrainte knapsack et ses éléments appartiennent à des ensembles D_k deux à deux disjoints. L'inégalité qu'induit C dans ce cas est appelée inégalité GUB-couverture.

Méthode du lifting

La méthode du "lifting" a été suggérée entre autres par Padberg [63], Wolsey [83] et Zemel [84]. Elle a été ensuite largement utilisée dans des implémentations d'algorithmes de plans coupants notamment par Crowder et al [16], Roy et Wolsey [70] et Bonami [8]. L'idée principale de cette approche consiste à projeter initialement le polyèdre pour lequel on veut dériver des facettes dans un espace de dimension inférieure en fixant quelques variables. Ensuite, on obtient une inégalité qui est une facette pour le polyèdre obtenu. Finalement, on dérive une facette du problème original en introduisant les variables fixées dans l'inégalité obtenue à l'étape précédente. Les variables peuvent être introduites d'une façon séquentielle (une seule à la fois) ou simultanément (par groupes). Cependant, le lifting séquentiel est plus utilisé vu qu'introduire les variables simultanément est généralement très couteux en terme de temps. L'exemple qui suit illustre la méthode du lifting séquentiel.

Exemple IV.4. *Soit la contrainte du sac-à-dos à variables binaires suivante*

$$x_1 + x_2 + 2x_3 + 2x4 + 3x5 \leq 3 \tag{IV.3}$$

Nous considérons l'ensemble $C = \{1, 2, 3\}$ et nous fixons les variables x_4 et x_5 à 0.
La contrainte de couverture s'écrit de la façon suivante :

$$x_1 + x_2 + x_3 \leq 2 \tag{IV.4}$$

Nous liftons la variable x_4, en la fixant à 1. L'inégalité (IV.4) s'écrit sous la forme :

$$x_1 + x_2 + x_3 + \alpha_4 x_4 \leq 2,$$

où α_4 représente le coefficient du lifting de la variable x_4 et l'inégalité (IV.3) devient

$$x_1 + x_2 + 2x_3 \leq 1.$$

Cette dernière est vérifiée si $x_1 = 1$ ou $x_2 = 1$, donc la valeur maximale que peut atteindre α_4 est 1. La contrainte de couverture (IV.4) devient alors :

$$x_1 + x_2 + x_3 + x_4 \leq 2.$$

De la même façon, nous liftons la variables x_5 en la fixant à 1. L'inégalité (IV.4) s'écrit sous la forme :

$$x_1 + x_2 + x_3 + x_4 + \alpha_5 x_5 \leq 2,$$

où α_5 représente le coefficient du lifting de la variable x_5. La contrainte (IV.3) s'écrit sous la forme suivante :

$$x_1 + x_2 + 2x_3 + 2x4 \leq 0,$$

qui est vérifiée lorsque $x_1 = x_2 = x_3 = x_4 = 0$ et donc la valeur maximale que peut atteindre α_5 vaut 2.
La contrainte de couverture obtenue s'écrit finalement sous la forme suivante :

$$x_1 + x_2 + x_3 + x_4 + 2x_5 \leq 2.$$

IV.2.2 Inégalités de couverture locales

Lorsque l'ensemble C est minimal, la contrainte de couverture (IV.2) obtenue est affermie. De plus, elle peut être étendue de telle sorte à obtenir la contrainte valide suivante :

$$\sum_{j \in E(C)} x_j \leq |C| - 1, \tag{IV.5}$$

où $E(C) = C \cup \{i \in 1, \ldots, n\}$ tels que $a_i \geq a_j$, $\forall j \in C$.

Exemple IV.5. *La contrainte de couverture* $x_3 + x_4 + x_5 + x_6 \leq 3$ *est dominée par la contrainte de couverture étendue* $x_1 + x_2 + x_3 + x_4 + x_5 + x_6 \leq 3$.

Notons que les inégalités (IV.2) et (IV.5) ne représentent pas des facettes pour Q_i. Cependant, des inégalités de couverture étendues peuvent être des facettes pour Q_i sous certaines conditions. En particulier, si l'ensemble C est partitionné en deux sous-ensembles C_1 et C_2 tels que $C_1 \neq \emptyset$, alors les inégalités de couverture étendues sont des facettes de :

$$N(C_1, C_2) = \{x \in \{0,1\}^n \, : \, \sum_{j=1}^{n} a_{ij} \leq b_i, x_i = 0 \, \forall i \notin C; \; x_i = 1 \, \forall i \in C_2\}.$$

Ce résultat permet de déduire qu'en liftant l'inégalité de couverture sur C_2 et $N \setminus C$, on obtient une facette du polytope Q_i.

La méthode du "lifting" peut être décrite par les étapes suivantes :

1. Considérer la face, notée $\mathcal{F}$, du polytope du sac-à-dos Q_i définie par les équations $x_j = 0, \; \forall j \in N \setminus C$. Alors, la contrainte (IV.2) est une facette de $\mathcal{F}$.
2. Effectuer des rotations de la contrainte (IV.2) pour en faire une facette de Q_i.

De façon analogue, la procédure du "lifting" peut être utilisée sur la face $\mathcal{F}'$ de Q_i définie par les équations de la forme $x_j = 1$. Dans ce cas, la contrainte

$$\sum_{j \in C \setminus C_2} x_j \leq |C \setminus C_2| - 1$$

est valide pour le polytope

$$\text{conv}\Big\{x \in \{0,1\}^{|N \setminus C_2|} \, : \, \sum_{j \in N \setminus C_2} a_{ij} x_j \leq b_i - \sum_{j \in C_2} a_{ij}\Big\}.$$

Le lifting des variables appartenant à $N \setminus C$ conduit à l'équation étendue suivante :

$$\sum_{j \in C \setminus C_2} x_j + \sum_{j \in N \setminus C} \alpha_j x_j \leq |C \setminus C_2| - 1, \tag{IV.6}$$

où chaque coefficient α_j du "lifting" satisfait la condition suivante : $0 \leq \alpha_j \leq |C \setminus C_2| - 1$. L'inégalité (IV.6) est une facette de $\mathcal{F}'$. Elle peut être liftée à son tour pour obtenir une facette du polytope Q_i de la forme :

$$\sum_{j \in C \setminus C_2} x_j + \sum_{j \in N \setminus C} \alpha_j x_j + \sum_{j \in C_2} \beta_j x_j \leq |C \setminus C_2| + \sum_{j \in C_2} \beta_j - 1.$$

Cette derniére contrainte est appelée *contrainte de couverture locale* (que l'on notera LCI). En général, le processus du calcul séquentiel des coefficients du lifting est appelé "down-lifting" lorsqu'il s'agit des variables fixées à 1 (appartenant à C_2). Par ailleurs, il est appelé "up-lifting" lorsqu'il s'agit des variables fixées à 0 (appartenant à $N \setminus C$). Dans le cas où $C_2 = \emptyset$, alors les contraintes LCI sont dites *simples*.

Notons que le calcul des coefficients du lifting revient à résoudre un problème d'optimisation comme sera décrit dans ce qui suit.

IV.2.2.1 Calcul des coefficients du lifting

Notons par x^* la solution fractionnaire du problème traité et soit $\alpha^T x \leq \beta$ l'inégalité valide à étendre. Supposons qu'on veut lifter la variable $x_k^* = 0$ (le cas $x_k^* = 1$ peut être traité d'une façon similaire). Ceci revient à trouver une inégalité valide pour

$$Q_i \cap \{x \,:\, x_j = x_j^* \; j \in N \setminus \{k\}\}$$

de dimension supérieure en résolvant le problème suivant :

$$(P1) \begin{cases} \max \; \lambda \\ \text{s.c.} \\ \qquad \alpha^T x + \lambda \leq \beta \quad \forall x \in Q_i \cap \{x \,:\, x_k = 1\}, \end{cases}$$

où α^T et β sont indépendants de λ. De plus, la valeur maximale de λ est déterminée en résolvant le problème :

$$(P2) \begin{cases} \max \; \alpha^T x \\ \text{s.c.} \\ \qquad x \in Q_i \cap \{x \,:\, x_k = 1\}. \end{cases}$$

Étant donnée $\bar{x}$ la solution optimale du problème (P2), alors la solution optimale de (P1) est donnée par $\bar{\lambda} = \beta - \alpha^T \bar{x}$.

Exemple IV.6. *Considérons la contrainte du sac-à-dos à variables binaires suivante :*

$$13x_1 + 7x_2 + 6x_3 + 5x_4 + 3x_5 + 10x_6 \leq 22,$$

et soit $x^* = (0, 0.4, 0.5, 0.5, 0.7, 1)$ *un point fractionnaire.*

L'ensemble $C = \{3, 4, 5, 6\}$ *est une couverture minimale. Nous considérons l'ensemble* C_2 *de* C *tel que* $C_2 = \{6\}$.

La containte de couverture $x_3 + x_4 + x_5 \leq 2$ *est valide pour le polytope restreint suivant :*

$$conv\Big\{x \in \{0,1\}^3 : 6x_3 + 5x_4 + 3x_5 \leq 12\Big\}.$$

Nous allons utiliser le lifting sequentiel $\{2, 6, 1\}$.

Étape 1 : *fixer* x_2 *à 1 ("up-lifting de* x_2*")*

Nous cherchons à determiner le coefficient α_2 *tel que :* $\alpha_2 x_2 + x_3 + x_4 + x_5 \leq 2$ *en résolvant le problème de sac à dos suivant :*

$$\max\Big\{x_3 + x_4 + x_5 : 6x_3 + 5x_4 + 3x_5 \leq 5\Big\}.$$

Comme la solution optimale de ce problème est égale à 1, $\alpha_2 = 2 - 1 = 1$, *donc la contrainte de couverture devient :*

$$x_2 + x_3 + x_4 + x_5 \leq 2.$$

Étape 2 : *fixer* x_6 *à 0 ("down-lifting de* x_6*").*

Nous cherchons le coefficient β_6 *tel que :* $x_2 + x_3 + x_4 + x_5 + \beta_6 x_6 \leq 2 + \beta_6$ *en résolvant le problème :* $\max\Big\{x_2 + x_3 + x_4 + x_5 : 7x_2 + 6x_3 + 5x_4 + 3x_5 \leq 22\Big\}$.

L'optimum vaut 4 et donc $\beta_6 = 4 - 2 = 2$. *Dans ce cas, la contrainte de couverture s'écrit sous la forme suivante :*

$$x_2 + x_3 + x_4 + x_5 + 2x_6 \leq 4.$$

Étape 3 : *fixer* x_1 *à 1 ("up-lifting de* x_1*")*

Nous cherchons le coefficient α_1 *verifiant la contrainte :* $\alpha_1 x_1 + x_2 + x_3 + x_4 + x_5 + 2x_6 \leq 4$.

La solution optimale du problème :

$$\max\Big\{x_2 + x_3 + x_4 + x_5 + x_6 : 7x_2 + 6x_3 + 5x_4 + 3x_5 \leq 9\Big\}$$

vaut 1 ; on en déduit que $\alpha_1 = 4 - 2 = 2$.

Finalement, la contrainte de couverture suivante :

$$2x_1 + x_2 + x_3 + x_4 + x_5 + 2x_6 \leq 4$$

coupe le point x^*.

IV.2.3 Inégalités de couverture globales

L'idée d'utiliser les facettes du polytope du sac-à-dos dans le traitement des problèmes plus complexes en variables binaires provient de Crowder et al [16]. Les auteurs ont démontré que l'intersection des Q_i peut donner une bonne approximation de P_I.
Cependant, pour plusieurs problèmes en variables binaires, notamment lorsque la matrice des contraintes est dense, l'utilisation des contraintes de couverture locales n'est pas toujours lucrative. En effet, il devient nécessaire dans certains cas de dériver des contraintes valides qui prennent en considération la structure globale du problème. Kontantinos et Letchford [50] ont proposé des contraintes de couverture globales pour résoudre le problème de sac-à-dos multidimensionnel (MDKP) en variables binaires.
Étant donnée une contrainte de couverture valide pour P_I et un sous ensemble $C_2 \subset C$, la contrainte :

$$\sum_{j \in C \setminus C_2} x_j + \sum_{j \in N \setminus C} \alpha_j x_j + \sum_{j \in C_2} \beta_j x_j \leq |C \setminus C_2| + \sum_{j \in C_2} \beta_j - 1.$$

est dite *contrainte de couverture globale* si elle est valide pour P_I. Cette contrainte est dite simple si $C_2 = \emptyset$.
Le lifting se fait de la même façon que pour les contraintes LCI et LGCI en remplaçant Q_i par P_I. Lifter la variable $x_k^* = 0$, par exemple, revient à trouver une inégalité valide pour $P_I \cap \{x : x_j = x_j^* \; j \in N \setminus \{k\}\}$ en résolvant le problème suivant :

$$(P1') \begin{cases} \max \; \lambda \\ \text{s.c.} \\ \qquad \alpha^T x + \lambda \leq \beta \quad \forall x \in P_I \cap \{x : x_k = 1\}. \end{cases}$$

Le λ maximal est déterminé en résolvant le problème suivant :

$$(P2') \begin{cases} \max \; \alpha^T x \\ \text{s.c.} \\ \qquad x \in P_I \cap \{x : x_k = 1\}. \end{cases}$$

Dans ce cas, le coefficient du lifting $\bar{\lambda}$ vaut $\beta - \alpha^T \bar{x}$.

IV.3 Contraintes valides pour le MMKP

Nous rappelons qu'une instance du MMKP est caractérisée par un ensemble S d'éléments répartis sur n classes $S_i,\ i \in I = \{1, \ldots, n\}$, disjointes ainsi qu'un vecteur de capacité $R = (R^1, R^2, \ldots, R^m)$. De plus, chaque classe $i,\ i \in I$, contient $r_i = |S_i|$ objets et à chaque objet j de la classe i est associé un profit positif v_{ij} et un vecteur poids $W_{ij} = (w_{ij}^1, \ldots, w_{ij}^2, \ldots, w_{ij}^m)$. L'objectif du problème est (i) de déterminer le sous-ensemble d'éléments satisfaisant les différentes contraintes de capacité et (ii) pour chaque classe un et un seul élément est choisi. Le sous-ensemble d'éléments sélectionné est choisi de manière à maximiser la valeur de la fonction objectif qui réalise le maximum de la somme de leurs profits.

Formellement, le MMKP peut s'écrire sous la forme suivante :

$$(MMKP)\begin{cases} Z(x) = \max \displaystyle\sum_{i=1}^{n}\sum_{j\in S_i} v_{ij}x_{ij} & & \\ \text{s.c.} \displaystyle\sum_{i=1}^{n}\sum_{j\in S_i} w_{ij}^k x_{ij} \leq R^k, & k \in \{1,\ldots,m\} & (1) \\ \displaystyle\sum_{j\in S_i} x_{ij} = 1, & i \in \{1,\ldots,n\} & (2) \\ x_{ij} \in \{0,1\}, & i \in \{1,\ldots,n\},\ j \in S_i. & \end{cases}$$

Dans cette partie, nous proposons des contraintes valides pour le MMKP. Ces contraintes sont construites en deux étapes :

Etape 1. Construire une couverture de cardinalité n dont les éléments appartiennent à des classes disjointes (appelée GUB-couverture).

Etape 2. Calculer les coefficients du lifting en évitant la résolution des problèmes d'optimisation.

Dans le résultat suivant, nous proposons une contrainte valide pour le problème MMKP.

Théorème IV.7. *Soit $C = \{ij\ :\ i \in I, j \in S_i\}$ un ensemble de $C \subseteq I \times S$ de cardinalité n tel que $\forall\ ij, i'j' \in C, i \neq i'$ et vérifiant :*

$$\sum_{ij\in C} w_{ij}^k > R^k,\ k \in \{1,\ldots,m\}$$

Alors la contrainte

$$\sum_{ij\in C} w_{ij}^k x_{ij} + \sum_{i\in I} \Big(Max(0, R^k - \sum_{i'j\in C, i'\neq i} w_{i'j}^k) \Big) \sum_{j'\in S_i, j'\neq j} x_{ij'} \leq R^k \qquad \text{(IV.7)}$$

est valide pour le MMKP.

Preuve : On peut remarquer que la contrainte (IV.7) est valide pour le MMKP vu que les coefficients du lifting sans calculés de telle sorte que la capacité R^k n'est jamais violée par les solutions du MMKP.

□

Notons que pour que la contrainte ne soit pas triviale, il faut qu'il existe au moins un élément $tt' \in C$ tel que :

$$\sum_{ij\in C-\{tt'\}} w_{ij}^k < R^k.$$

Notons aussi que la contrainte (IV.7) devient plus intéressente lorsque l'ensemble C est minimal. Dans ce cas, elle peut s'écrire sous la forme suivante :

$$\sum_{ij\in C} w_{ij}^k x_{ij} + \sum_{i\in I} \Big(R^k - \sum_{i'j\in C, i'\neq i} w_{i'j}^k \Big) \sum_{j'\in S_i, j'\neq j} x_{ij'} \leq R^k \qquad \text{(IV.8)}$$

Dans la suite, nous utilisons les contraintes (IV.8) que nous notons (VLI).

IV.4 Inégalités de couverture étendues pour le MMKP

Afin de résoudre le MMKP, nous nous intéressons plutôt aux contraintes GUB-couverture locales, notées LGCI. Ce choix a été motivé par la particularité de la structure du problème (c-à-d les contraintes de choix sur les classes). Les inégalités LGCI sont obtenues de la même façon que les inégalités LCI en construisant tout d'abord une GUB-couverture C, puis, en liftant les variables. Ensuite, en s'appuyant sur le principe des inégalités de couverture globales, nous proposons aussi des inégalités GUB-couverture globales pour le MMKP, que l'on notera GLGCI. Notons que ces contraintes généralisent les inégalités GUB-couverture présentées dans Gu et al [36], puisque toutes les contraintes du problème sont prises en compte. Finalement, nous proposons une technique pour construire une GUB-couverture initiale pour le MMKP.

Entrée : Une solution factionnaire x^* et une contrainte knapsack k du MMKP.

Sortie : Une GUB-couverture minimale pour le MMKP.

1. Initiation :

   Poser $C = \emptyset$;

2. **Pour** chaque classe i, $i \in \{1, \ldots, n\}$, **faire**

   (a) Ordonner les éléments de la solution x^*_{ij} $(j = 1, \ldots, r_i)$ par ordre décroissant.

   (b) Construire l'ensemble G_i tel que $G_i = \{ij \ : \ x^*_{ij} > 0\}$.

   (c) Soit z^i_1 le premier élément de l'ensemble G_i, alors poser $C := C \cup \{z^i_1\}$.

3. **Si** $\sum_{ij \in C} w^k_{ij} > R^k$ et C est une couverture minimale, aller à l'étape 4.
   Sinon effectuer des permutations dans les ensembles G_i permettant la construction d'une couverture minimale ;

4. **Fin.**

Fig. IV.1 : Construction d'une GUB-couverture

IV.4.1 Construction d'une couverture minimale pour le MMKP

Soit x^* la solution de la relaxation continue du MMKP et soit $k \in \{1, \ldots, m\}$ l'indice d'une contrainte knapsack. Pour déterminer une GUB couverture, notée C, nous appliquons les étapes présentées dans la figure IV.1. En effet, nous construisons C en choisissant de chaque classe i l'élément réalisant la valeur maximale de x^*. Dans le cas où l'ensemble C ainsi construit n'est pas une couverture minimale (ou simplement une couverture), alors nous effectuons des permutations au niveau des éléments choisis de chaque classe jusqu'à obtention d'une couverture minimale.

IV.4.2 Problème de séparation

Supposons que l'on dispose d'une solution fractionnaire x^*. Pour le MMKP, nous pouvons adapter l'heuristique de Gu et al [36] pour (i) générer des contraintes CGCI et GCGCI violées et (ii) des contraintes VLI. En effet, nous pouvons procéder de la façon suivante :

1. Considérer la solution x^* du problème courant. Soient $N^0 = \{j \in S \ : \ x^*_j = 0\}$ et $N^1 = \{j \in S \ : \ x^*_j > 0\}$.

2. Pour $k = 1, \ldots, m$ faire

   (a) *Couverture initiale :*

   construire une GUB-couverture minimale C en appliquant l'heuristique décrite dans la figure IV.1 .

   (b) *Partition de la couverture :*

   diviser la couverture C en deux ensembles disjoints : $C_2 = \{j \in C \ : \ x_j^* = 1\}$ et $C_1 = C \setminus C_2$.

   (c) *Opération du Lifting :*

   Cette phase comporte trois étapes principales : lifter les variables de $N^1 \setminus C$, puis les variables de C_2 et enfin les variables de $N^0 \setminus C$.

   Nous procédons de la façon suivante :

   i. Construire la couverture $\sum_{j \in C \setminus C_2} x_j \leq |C \setminus C_2| - 1$, qui est valide pour le polytope restreint.

   ii. Construire les ensembles $G_i \, (i = 1, \ldots, n)$ tels que $G_i = \{ij \ : \ x_{ij}^* > 0\}$. Puis, ordonner les variables de chaque ensemble G_i par ordre décroissant. Ensuite, lifter ("up-lifting") les variables suivant l'ordre des indices des ensembles G_i.

   iii. **Si** la contrainte $\sum_{j \in C_1} x_j^* + \sum_{j \in N^1 \setminus C} \alpha_j x_j^* \leq |C_1| - 1$ n'est pas violée, alors poser $k = k + 1$ et aller à 2a.

   iv. Lifter ("down-lifting") les variables de C_2 dans l'ordre de leurs indices.

   v. Lifter ("up-lifting") les variables de $N^0 \setminus C$ dans l'ordre des ensembles G_i et des indices des variables de chacun d'entre eux.

   (d) Construire une inégalié VLI en utilisant la GUB-couverture C comme couverture initiale et tester si elle est violée.

   (e) retourner la couverture violée LGCI (ou GLGCI) et la contrainte violée VLI.

IV.5 Un algorithme de branch-and-cut pour le MMKP

En général, un algorithme de type branch-and-cut suit le principe des algorithmes par séparation et évaluation. Cependant, l'ingrédient principal de la méthode générique

s'articule autour de points suivants :

1. Injection des contraintes valides dans le problème original (ou au problème en cours de résolution).
2. Introduction des coupes lorsque ces dernières sont construites.
3. Gestion de ces contraintes/coupes dans une méthode de séparation et d'évaluation.

Plus précisément, notre méthode s'appuie sur les points (1), (2) et (3) précédemment cités avec une éventuelle réorganisation afin d'évaluer la pertinence de chacune des contraintes introduites dans cette étude (c-à-d les contraintes VLI et LGCI (ou GLGCI)). Notons aussi que l'existence d'un solveur performant, comme le Cplex, permet de faciliter la tâche de l'implémentation d'un branch-and-cut, puisque la souplesse de ce dernier permet d'exploiter certaines fonctions propres au solveur pour la construction d'une méthode de type branch-and-cut.

Dans une étude préliminaire (que nous détaillons dans la Section IV.6), nous avons considéré deux alternatives pour la résolution du problème MMKP :

- La première alternative consiste à tester un "cut-and-branch". Cette approche consiste à introduire des contraintes, présentées dans cette étude, dans le premier nœud (c-à-d le niveau supérieur de l'arborescence de recherche), avant de lancer le solveur Cplex.
- La deuxième alternative consiste à introduire d'une façon successive ces contraintes sur les 1000 premiers nœuds.

IV.6 Partie expérimentale

Dans cette section, nous évaluons la performance de l'algorithme proposé sur deux ensembles d'instances. Cet algorithme est codé en **C++**, il utilise les API du Cplex et il est exécuté sur une station SUN Ultra-Sparc10 (250 Mhz et avec 1Gb de mémoire RAM).

Dans un premier temps, nous testons un premier groupe d'instances composé de 13 instances (notées I01,...,I13). Ces instances sont extraites de Khan et al. [47] dont la taille varie selon le nombre de classes et le nombre d'éléments par classe. Notons que ces instances ont été également utilisées dans le chapitre 3 comme benchmark pour les méthodes heuristiques proposées.

Dans un deuxième temps nous menons notre étude expérimentale sur un deuxième ensemble d'instances générées aléatoirement. Chacune des instances utilisées est représentée par le nombre de classes m généré aléatoirement dans l'intervalle discret $\{50, 100, 200, 300, 350\}$, par le nombre d'éléments par classe fixé à 10 et par le nombre de contraintes de type knapsack fixé dans l'intervalle discret $\{8, 10\}$.

Afin de comparer la performance de notre algorithme par rapport à la performance du solveur Cplex ainsi que la méthode coopérative proposée dans le chapitre 3, nous avons limité le temps d'exécution à 3600 secondes. Cette limite sert principalement à arrêter l'exécution de l'algorithme si ce dernier n'arrive pas à résoudre à l'optimum l'instance du problème MMKP. D'une part, cette limite reste raisonnable pour une méthode exacte si cette dernière arrive à résoudre le problème à l'optimum. D'autre part, une telle limite nous permet de discuter par la suite la pertinence des contraintes ajoutées au problème MMKP.

IV.6.1 Le premier groupe d'instances

Le premier groupe d'instances, composé de 13 instances, représente les instances de Khan et al. [47]. Il est composé de six instances de taille moyenne dont les solutions optimales sont connues et de sept autres instances de grande taille dont les solutions optimales n'ont pas été prouvées.

Cette section est composée de deux parties. Dans la première partie nous proposons d'étudier la pertinence des contraintes proposées, en particulier, sur les six premières instances dont les solutions optimales sont connues. Dans la deuxième partie nous élargissons l'étude pour évaluer la performance de l'algorithme de branch and cut, noté BC, lorsque les différentes contraintes/coupes sont introduites.

Plus précisément, dans la première partie nous comparons les performances du solveur Cplex sans aucune contrainte, le soveur Cplex en injectant les contrantes VLI à la racine de l'arborescence de recherche ainsi que les trois versions de l'algorithme BC (avec les contraintes VLI, avec les contraintes LGCI et VLI, et avec les contraintes GLGCI et VLI).

La Table IV.1 résume les résultats obtenus pour chacune des versions de l'algorithme considéré (c-à-d le Cplex et le BC). La colonne 1 de la table reporte le nom de l'instance

	Cplex		Cplex−VLI		BC−VLI		BC−LGCI & VLI		BC−GLGCI & VLI	
Inst	cpu	Nœuds	cpu	Nœuds	cpu	Nœuds	cpu	Nœuds	cpu	Nœuds
I01	0.01	0	0.02	0	0.01	0	0.02	0	0.01	0
I02	0.02	0	0.01	0	0.01	0	0.01	0	0.01	0
I03	8.2	8065	6.97	6803	5.83	5761	7.21	6021	5.01	4570
I04	175.15	123519	151.14	110580	132.76	101632	120.1	92853	113.1	80754
I05	0.05	0	0.06	0	0.07	2	0.06	0	0.05	0
I06	0.14	14	0.17	14	0.16	11	0.1	8	0.21	7
Moyenne	30.60	21933.00	26.40	19566.17	23.14	17901.00	21.25	16480.33	19.73	14221.83

Tab. IV.1 : Comportement du solveur Cplex ainsi que l'algorithme BC lors de l'introduction des différentes contraintes/coupes

traitée. La colonne 2 (resp. colonne 3) reporte le temps d'exécution (resp. le nombre de nœuds) que nécessite (resp. que génère) le solveur Cplex pour résoudre l'instance traitée à l'optimum. La colonne 4 (resp. colonne 5) reporte le temps d'exécution (resp. le nombre de nœuds) que nécessite (resp. génère) la version de l'algorithme combinant le Cplex et les contraintes de type VLI introduites à la racine. Les temps d'exécution et le nombre de nœuds générés par les trois versions de l'algorithme BC sont représentés par les contenus des colonnes 6 à 11.

A la suite de ces résultats, nous pouvons remarquer que :

1. L'introduction des contraintes VLI à la racine permet d'accélérer la résolution de certaines instances. En effet, ce phénomène peut se justifier par le temps d'exécution (resp. le nombre de nœuds) moyen qui passe de 30.60 secondes (resp. 21933.00) pour le Cplex à 26.40 secondes (resp. 19566.17) pour la version améliorée.
2. La première version de l'algorithme proposé BC−VLI a un bon comportement, puisqu'il permet de réduire le temps d'exécution moyen (23.14 secondes). Le nombre moyen de nœuds générés décroît et il vaut dans ce cas 17901.00 nœuds.
3. La combinaison des contraintes LGCI et VLI fait augmenter la performance de l'algorithme BC. En effet, cette fois-çi le temps d'exécution décroît (qui vaut 21.25 secondes en moyenne) et le nombre moyen de nœuds générés diminue (il passe à 16480.33 nœuds en moyenne).
4. La dernière version de l'algorithme BC, qui combine les contraintes GLGCI et VLI, peut être considérée come la version la plus performante sur ce jet d'instances. En effet, dans ce cas le temps d'exécution moyen vaut 19.73 secondes et le nombre

moyen de nœuds générés passe à 14221.83 nœuds.

5. Finalement, une comparaison directe avec le solveur Cplex montre que la dernière version de l'algorithme BC permet une amélioration moyene de 35.51% du temps d'exécution et permet une réduction de 35.16% de nœuds en moyenne.

Dans la deuxième partie de cette section, nous comparons la performance de différentes versions de l'algorithme BC par rapport à la performance du solveur Cplex ainsi que la méthode coopérative proposée dans le Chapitre 3. Nous rappelons l'existence d'une limite sur le temps d'exécution (fixée à 3600 secondes), qui correspond au cas où l'algorithme considéré n'arrive pas à résoudre l'instance à l'optimum. Les solutions fournies par chacune des versions de l'algorithme proposé sont comparées aux solutions obtenues par les approches suivantes :

- La procédure d'arrondi généralisée, que nous avons noté PAG.
- Le solveur Cplex, limité aussi à 3600 secondes.
- Les meilleures bornes inférieures connues jusqu'à présent pour les instances non résolues à l'optimum.

Nous rappelons que trois versions de l'algorithme BC sont considérées. Dans la première version de l'algorithme, seules les contraintes valides VLI sont injectées. Dans la deuxième version seules les contraintes de couverture LGCI sont introduites. Finalement dans la troisième version de l'algorithme, les contraintes de couverture GLGCI sont utilisées.

La performance des différents algorithmes testés est représentée par la Table IV.2. Dans cette table, le nom de chaque instance considérée est représenté dans la colonne 1. Les meilleures solutions (bornes inférieures ou solutions optimales) connues et extraites du Chapitre 3 sont reportées dans la colonne 2. La colonne 3 contient les solutions obtenues par la procédure d'arrondi généralisée PAG. Les solutions obtenues par le solveur Cplex ainsi que le temps d'exécution que nécessite ce solveur sont respectivement représentés par le contenu des colonnes 4 et 5. Les résultats des trois versions de l'algorithme BC sont reportés dans les colonnes 6, 8 et 10 correspondant, respectivement, à l'utilisation des contraintes VLI, LGCI et GLGCI. Le temps d'exécution que nécessite chacune des versions de l'algorithme BC est représenté dans les colonnes 7, 9 et 11, respectivement. Notons que le symbole "-" signifie que l'algorithme atteint le temps limite que nous avons

			Cplex		BC		BC		BC	
Inst	Meil/Opt Sol	Sol. PAG	Sol.	cpu	VLI	cpu	LGCI & VLI	cpu	GLGCI & VLI	cpu
I01	173^o	173	173		173		173		173	
I02	364^o	364	364		364		364		364	
I03	1602^o	1602	1602		1602		1602		1602	
I04	3597^o	3597	3597		3597		3597		3597	
I05	3905.7^o	3905.7	3905.7		3905.7		3905.7		3905.7	
I06	4799.3^o	4799.3	4799.3		4799.3		4799.3		4799.3	
I07	24587	24587	24584	-	24584	-	24584	-	24586*	-
I08	36894	36869	36869	-	36871*	-	36871*	-	36871*	-
I09	49179	49175	49155	-	49155	-	49157*	-	49161*	-
I10	61464	61461	61446	-	61446	-	61446	-	61447*	-
I11	73783	73759	73759	-	73759	-	73759	-	73759	-
I12	86080	86071	86071	-	86071	-	86071	-	86071	-
I13	98438	98409	98418	-	98418	-	98418	-	98418	-

Tab. IV.2 : Comportement de l'algorithme BC selon les contraintes/coupes appliquées

fixé à 3600 seconds, le symbole "o" signifie que la solution optimale a été prouvée et finalement le symbole "*" signifie que l'algorithme fournit une solution de meilleure qualité que le solveur Cplex, que l'algorithme PAG ou que les meilleures solutions connues.

A partir de la Table IV.2, nous pouvons remarquer que :

1. Les trois versions de l'algorithme BC ne sont pas capables de résoudre les instances I07 à I13. Au fait ces instances représentent une catégorie d'instances très ardues pour le MMKP.
2. Aucune des trois stratégies (les trois versions du BC) n'est capable d'atteindre toutes les meilleures solutions (bornes inférieures) de la colonne 2.
3. Une comparaison directe avec la méthode PAG montre que :
   (a) La première version du BC (utilisant les contraintes VLI) arrive à améliorer la solution de l'instance I08,
   (b) La deuxième version du BC améliore les solutions de deux instances ; une par rapport au solveur Cplex (instance I08) et une par rapport à GAP (instance I09).
   (c) Enfin, la troisième version du BC améliore quatre solutions, une solution par rapport à la méthode PAG (Instance I08) et trois solutions par rapport au solveur Cplex (instances I07, I09 et I10).

On peut conclure que les trois versions de l'algorithme BC restent intéressantes avec ces réglages lorsqu'il s'agit de traiter des instances de taille modérée, mais nous pensons qu'une étude plus poussée reste nécessaire pour palier le problème des instances de grande taille. De même, nous pensons aussi que transformer l'algorithme BC en une heuristique nécessite un peu plus de tests numériques.

IV.6.2 Le deuxième groupe d'instances

Dans cette section nous nous concentrons sur la performance de l'algorithme BC sur des instances de taille modérée. Nous évaluons donc la performance de l'algorithme sur un autre ensemble d'instances aléatoires de taille modérée. Ces instances sont générées comme suit. Le nombre de classes de ces instances est fixé dans l'intervalle discret $\{50, 100, 200, 300, 350\}$, le nombre d'éléments par classe est fixé à 10 et le nombre de contraintes par classe est fixé dans l'intervalle discret $\{8, 10\}$.

Nous présentons deux versions de l'algorithme BC. Dans un premier temps, nous utilisons une première version qui combine les contraintes VLI et les contraintes LGCI. Dans un deuxième temps, nous considérons une deuxième version dont laquelle les contraintes VLI et GLGCI sont combinées.

					Solveur Cplex		BC−LGCI & VLI		BC−GLGCI & VLI	
Inst	n	r_i	m	Opt	Nœuds	cpu.	Nœuds	cpu.	Nœuds	cpu
Ia1	50	10	5	12955	1218	2.29	1000	2.56	762	1.31
Ia2	50	10	8	13815	3131	5.62	1200	3.20	612	1.02
Ia3	50	10	8	12263	433300	664.849	374644	599.49	123691	257.35
Ia4	100	10	8	40061	623280	2548.22	517032	1918.31	454261	1870.02
Ia5	200	10	8	107098	123570	1621.53	92138	975.12	68520	700.63
Ib1	300	10	8	80201	215160	1842.33	73422	1007.29	98321	1322.1
Ib2	350	10	8	93636	40080	554.92	9690	192.23	10255	351.23
Ib3	350	10	8	93651	450621	3340.85	325298	1805.78	28100	1138.36
Ib4	350	10	8	89826	431000	4041.71	361552	4630.12	337290	3203.01
Moyenne					257928.89	1624.70	195108.44	1237.12	124645.78	982.78

Tab. IV.3 : Performance de l'algorithme BC

Les performances des deux versions de l'algorithme BC, sur l'ensemble de ces instances, sont reportées dans la Table IV.3. Dans cette dernière, nous représentons :

- le nombre de classes n (colonne 2) ;
- le nombre d'éléments r_i de chaque classe i, $i = 1, \ldots, n$ (colonne 3) ;

- le nombre de contraintes m de type knapsack (colonne 4) ;
- la solution optimale de l'instance traitée (colonne 5) ;
- le nombre de nœuds (Nœuds : colonne 6) développés par le solveur Cplex pour résoudre l'instance ainsi que le temps de calcul qu'il nécéssite (colonne 7) ;
- pour la première version de l'algorithme BC (en combinant les contraintes LGCI et VLI) : le nombre de nœuds (Nœuds : colonne 8) developpés et le temps de calcul (cpu : colonne 11) qu'il nécessite pour résoudre l'instance à l'optimum ;
- la deuxième version de l'algorithme BC (en combinant les contraintes GLGCI et VLI) : le nombre de nœuds (Nœuds : colonne 10) developpés et le temps de calcul (cpu : colonne 11) qu'il nécessite.

A partir de la Table IV.3, nous remarquons que :

1. Les deux versions de l'algorithme BC sont plus performantes que le solveur Cplex.
2. L'association des deux types de contraintes LGCI et VLI permet d'accélérer le processus de recherche. En effet, pour cette première version de l'algorithme BC, elle nécessite un temps d'exécution moyen de 1237.12 secondes (comparé au 1624.70 secondes que nécessite le solveur Cplex). En d'autre termes, la première version du BC réalise une amélioration moyene de 23.86% du temps d'exécution et permet une réduction moyenne de 24.36% sur le nombre de nœuds générés.
3. La deuxième version du BC (combinant les contraintes GLGCI et VLI) est encore plus performante. En effet, dans ce cas le temps d'exécution moyen diminue (il vaut 982.78 seconds) et le nombre moyen de nœuds générés est réduit (il vaut 124645.78). En d'autres termes, le temps d'exécution moyen est amélioré de 39.51% et le nombre moyen de nœuds générés est réduit de 51.67%.

IV.7 Conclusion

Dans ce chapitre, nous avons proposé une méthode de résolution exacte pour le problème du sac-à-dos généralisé à choix multiple. Cette méthode s'appuie sur le principe d'un branch-and-cut, une extension d'une méthode par séparation et évaluation. Dans un premier temps, nous avons proposé des contraintes valides pour le problème. Dans un deuxième temps, nous avons proposé une généralisation des contraintes GUB-couverture

pour le problème en prenant en considération toute la matrice des contraintes lors du calcul des coefficients du lifting. Ensuite, ces différentes contraintes ont été introduites dans un schéma de séparation et d'évaluation. Finalement, afin de tester la performance de la méthode proposée, nous avons mené une première étude expérimentale sur deux groupes d'instances : un premier groupe composé d'instances de la littérature et un deuxième groupe d'instances générées aléatoirement. Dans cette étude expérimentale, nous avons discuté la performance d'une telle approche sur des instances de taille modérée ainsi que ses inconvénients sur des instances de grande taille.

Conclusion générale et perspectives

Dans cette thèse, nous avons étudié le problème du knapsack généralisé à choix multiple. Notre travail s'est focalisé autour de deux différentes approches de résolution qui sont souvent complémentaires : les méthodes approchées ou heuristiques et les méthodes exactes.

Dans la première partie, nous nous sommes intéréssés à la résolution approchée du problème en se basant sur les méthodes d'arrondi complétées par des méthodes de voisinage. Nous nous sommes aussi appuyés sur la méthode de génération de colonnes qui nous permettait d'accélérer le processus de recherche et d'atteindre des segments intéressants de l'espace de recherche. Nous avons développé deux approches principales. Concernant la première approche, nous avons présenté deux versions évolutives. Dans un premier temps, nous avons proposé une première version de l'algorithme qui s'appuie sur une procédure d'arrondi. Ensuite, nous avons proposé une deuxiéme version de l'algorithme dont l'idée principale reposait sur la complémentarité entre le traitement effectué par la méthode d'arrondi et celui d'un algorithme exact. Cette version de l'algorithme s'appuie sur deux phases : (i) la première phase consiste à appliquer, de manière itérative, la génération de colonnes et arrondir la solution optimale de la relaxation continue obtenue pour fixer une partie des variables et (ii) la deuxième phase résout de façon exacte la partie restreinte du problème. Ces deux versions de l'algorithme ont permis, dans un premier temps, d'améliorer la qualité d'un nombre de solutions de la littérature.
Dans la même partie, nous avons proposé une extension des deux versions de l'algorithme précédemment présentées. L'idée principale s'appuie sur l'hybridation de chaque version de l'algorithme avec une méthode par séparation et évaluation restreinte. Dans une partie expérimentale, nous avons évalué la performance des deux nouvelles versions de l'algorithme sur différentes instances ardues de la littérature. Dans cette partie, nous avons

souligné l'efficacité des deux versions à produire de bonnes solutions et la rapidité de la première version. Nous avons aussi constaté que la deuxième version donnait de meilleures solutions puisqu'elle arrivait à améliorer un certain nombre de solutions de la littérature.

La deuxième approche que nous avons proposée se base sur les méthodes de voisinage. Elle conjugue les efforts de la méthode de branchement local standard et la génération de colonnes pour une meilleure exploration de l'espace de recherche. Elle utilise une procédure d'arrondi hybride pour explorer les voisinages. Ensuite, une version augmentée de l'approche a été également proposée. Elle consiste, de façon analogue à la première approche, à développer un algorithme par séparation et évaluation tronqué où la première version de l'approche est appliquée à un sous-ensemble de nœuds élites. Dans une partie expérimentale, nous avons constaté que le branchement local standard restait compétitif comparé aux solutions des instaces de la littérature. La première version tire profit de la génération de colonnes et arrive à améliorer la plupart des instances. La deuxième domine les deux autres versions vu la qualité des solutions qu'elle fournit mais nécessite plus de temps de calcul.

Dans la deuxième partie de cette thèse, nos travaux ont porté sur la résolution exacte du problème du knapsack multidimensionnel à choix multiple. Nous avons proposé une méthode exacte qui s'appuie sur une procédure de "branch-and-cut". Le principe de la méthode s'appuie sur (i) la détermination de nouvelles contraintes valides et (ii) l'utilisation de contraintes de couverture locales et globales particulières. Nous avons ensuite présenté une étude expérimentale préliminaire de l'algorithme sur deux groupes d'instances : un groupe d'instances composé d'instances principalement ardues et un deuxième groupe d'instances composé d'instances aléatoires de taille modérée. Nous avons constaté que l'algorithme était très performant sur des instances de taille modérée, mais il restait impuissant face à des instances de grande taille.

Parmi les perspectives à court terme, il nous semble important de porter des amélioration sur l'algorithme de branch-and-cut présenté. Nous pensons au problème de séparation, à l'utilisation d'autres coupes ainsi qu'au mécanisme de suppression des contraintes injectées. Une autre direction de recherche qui nous semble intéressante c'est de réfléchir sur l'hybridation entre un branch-and-cut et le branchement local utilisant la génération de colonnes

Parmi les perspectives à moyen terme, nous pensons à la parallélisation des méthodes hybrides que nous avons proposées. Le but est de pouvoir considérer plus de nœuds dans les versions augmentées que nous avons présenté et ainsi élargir le domaine de recherche des solutions. Une autres direction de recherche consiste à étendre l'application des méthodes de résolution hybrides mises en oeuvre pour le problème MMKP à d'autres problèmes de type knapsack.

Liste des tableaux

Table des figures

Bibliographie

[1] A. Atamtürk. Cover and pack inequalities for (mixed) integer programming. *Annals of Operations Research*, 139(1) :21–38, 2005.

[2] E. Balas. Facets of the knapsack polytope. *Mathematical Programming*, 8 :146–164, 1975.

[3] E. Balas, S. Ceria, G. Cornuéjols. A lift-and-project cutting plane algorithm for mixed 0-1 programs. *Mathematical Programming*, 58 :295–324, 1993.

[4] E. Balas, S. Ceria, G. Cornuéjols. Solving mixed 0-1 programs by a lift-and-project method. *In : Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 232–242, 1993.

[5] E. Balas, S. Ceria, G. Cornuéjols. *Mixed 0-1 programming by lift-and-project in a branch-and-cut framework.* Rapport technique, Management Science Research Report MSRR-603, GSIA, Carnegie Mellon University, Pittsburgh, 1994.

[6] C. Barnhart, E. L. Johnson, M. W. P. Savelsbergh and P. H. Vance. Branch and price : column generation for solving huge integer programs. *Operations Research*, 46(3) :316–329, 1998.

[7] J. F. Benders. Partitioning procedures for solving mixed variables programming problems. *Numerische Mathematik*, 4 :238–252, 1962.

[8] P. Bonami. *Étude et mise en œuvre d'approches polyèdriques pour la résolution de programmes en nombres entiers ou mixtes gńéraux.* PhD thesis, Université Paris 6, 2003.

[9] M. E. Bouzgarrou. *Parallélisation de la méthode du "branch-and-cut" pour résoudre le problème du voyageur de commerce.* PhD thesis, Institut National Polytechnique de Grenoble, 1998.

[10] V. BOYER. *Contribution à la programmation en nombre entier*. PhD thesis, L'Institut National des Sciences Appliquées de Toulouse, 2007.

[11] N. CHERFI and M. HIFI. Algorithms for multiple-choice multi-dimensional knapsack problems. In *Conférence internationale métaheuristique META'06, Hammamet, Tunisia, 2-4 November*, 2006.

[12] N. CHERFI and M. HIFI. A column generation method for the multiple-choice multi-dimensional knapsack problem. *Combinatorial Optimization and Applications, à paraître*, 2008.

[13] P. CHU and J. E. BEASLEY. A genetic algorithm for the multidimensional knapsack problem. *Journal of heuristics*, 4(1) :63–86, 1998.

[14] V. CHVÀTAL. Edmonds polytopes and weakly hamiltonian graphs. *Mathematical Programming*, 5 :29–40, 1973.

[15] V. CHVÀTAL. *Linear Programming*. W. H. Freeman and Company, 1983.

[16] H. CROWDER, E. JOHNSON and M. PADBERG. Solving large-scale 0-1 linear programming programms. *Operations Research*, 31, 1983.

[17] G. B. DANTZIG. Decomposition principle for linear programs. *Operations research*, 8 :101–111, 1960.

[18] G.B. DANTZIG. Discrete variable extremum problems. *Operations research*, 5 :266–277, 1957.

[19] G. B. DANTZIG, R. FULKERSON and S. JOHNSON. Solution of a large-scale traveling salesman problem. *Operations research*, 2 :393–410, 1954.

[20] G. Desaulniers, J. Desrosiers, M. M. Solomon. *Column Generation*. Springer, 2005.

[21] M. DESROCHERS, J. DESROSIERS and M. M. SOLOMON. A new optimization algorithm for the vehicle routing problem with time windows. *Operations Research*, 40(2) :342–354, 1992.

[22] K. DUDZINSKI and S. WALUKIEWICZ. Exact methods for the knapsack problem and its generalizations. *Mathematical Programming*, 29 :231–249, 1984.

[23] M. ELKIHEL. *Programmation dynamique et rotations de contraintes pour les problèmes d'optimisation entière*. PhD thesis, Université des Sciences et Techniques de Lille, 1984.

[24] D. Fayard and G. Plateau. An algorithm for the solution of the 0-1 knapsack problem. *Computing*, 28 :269–287, 1982.

[25] M. Fischetti and A. Lodi. Local branching. *Mathematical Programming*, 98 :23–47, 2003.

[26] M. Fischetti and P. Toth and D. Vigo. A branch-and-cut algorithm for the capacitated vehicle routing problem on directed graphs. *OR*, 42 :846–859, 1994.

[27] M. Fischetti, M. Polo and M. Scantamburlo. A local branching heuristic for mixed-integer programs with 2-level variables. *Networks*, 44 :61–72, 2004.

[28] P. C. Gilmore and R.E. Gomory. A linear programming approach to the cutting stock problem. *Operations Research*, 9 :849–858, 1961.

[29] P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting stock problem-part ii. *Operations Research*, 11 :863–888, 1963.

[30] P. C. Gilmore and R. E. Gomory. Multistage cutting-stock problems in two or more dimensions. *Operations Research*, 13 :94–120, 1965.

[31] D. Goldfarb and M. J. Todd. *Linear programming.* In M. J. Todd G. L. Nemhauser A.H.G. Rinnooy Kan, édieur, Handbooks in operations research and management science, volume 1, pages 73-170. Elsevier, 1989.

[32] R. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64 :275–278, 1958.

[33] H. Greenberg and R. L. Hegerich. A branch search algorithm for the knapsack problem. *Management Science*, 16 :327–332, 1970.

[34] M. Grötschel and M. Jünger and G. Reinelt. A cutting plane algorithm for the linear ordering problem. *Operations Research*, 32 :1195–1220, 1984.

[35] M. Grötschel, L. Lovàsz and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization.* 1988.

[36] Z. Gu, G.L. Nemheuser and M. W. P. Savelsbergh. Cover inequalities for 0-1 linear programs : computation. *INFORMS Journal on Computing*, 10 :427–437, 1998.

[37] R. P. Hernandez and N. J. Dimopoulos. A new heuristic for solving the multichoice multidimensional knapsack problem. *Systems, Man and Cybernetics, Part A, IEEE Transactions*, 35(5) :708–717, 2005.

[38] M. Hifi, M. Michrafy and A. Sbihi. Heuristic algorithms for the multiple-choice multidimensional knapsack problem. *Journal of the Operational Research Society*, 55 :1323–1332, 2004.

[39] M. Hifi, M. Michrafy and A. Sbihi. A reactive local search-based algorithm for the mutiple-choice multi-dimensional knapsack problem. *Computational Optimization and Applications*, 33 :271–285, 2006.

[40] M. Hifi, S. Sadfi and A. Sbihi. An exact algorithm for the multiple-choice multidimensional knapsack problem. *Cahiers de la MSE, CERMSEM, Maison des Sciences Economiques, Université Paris 1 Panthéon-Sorbonne*, (24), 2004.

[41] E. Horowitz and S. Sahni. Computiong partitions with applications to the knapsack problem. *Journal of ACM*, 21 :277–292, 1974.

[42] S. Joy and B. Borchers and J. E. Mitchell. A branch-and-cut algorithm for MAX-SAT and weighted MAX-SAT. In D. Du, J. Gu, and P. M. Pardalos, editors, *Satisfiability Problem : Theory and Applications*, volume 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 519–536. AMS/DIMACS, 1997.

[43] K. Karmarkar. A new polynomial time algorithm for linear programming. *Combinatorica*, 4 :373–395, 1984.

[44] H. Kellerer, U. Pferschy and D. Pisinger. *Knapsack Problems.* Springer Verlag, 2004.

[45] L. G. Khachian. A polynomial algorithm in linear programming. *Soviet Mathmatics Doklady*, 20 :191–194, 1979.

[46] S. Khan. *Quality adaptation in a multi-session adaptive multimedia system : model and architecture.* PhD thesis, Department of Electronical and Computer Engineering, Uiversity of Victoria, 1998.

[47] S. Khan, K. F. Li, E. G. Manning and MD. M. Akbar. Solving the knapsack problem for adaptive multimedia systems. *Studia Informatica, an International Journal, Special Issue on Cutting, Packing and Knapsacking problems*, 2/1 :154–174–589, 2002.

[48] V. Klee and G. J. Minty. How good is the simplex algorithm. *In Shisha Os, éditeur, Inequalities III, Academic Press, New-York*, pages 159–175, 1972.

[49] P. J. Kolesar. A branch and bound algorithm for the knapsack problem. *Management Science*, 13 :723–735, 1967.

[50] K. Konstantinos and N. Letchford. Local and global lifted cover inequalities for the 0-1 multidimensional knapsack problem. *European journal of operational research*, 186(1) :91–103, 2008.

[51] L. S. Lasdon. *Optimization theory for large systems.* Macmillan, New York.

[52] M. Lebbar. *Résolution de problèmes combinatoires dans l'industrie, apport de la programmation mathématique et des techniques de décomposition.* PhD thesis, École Centrale, Paris, 2000.

[53] M. Magazine and O. Oguz. A heuristic algorithm for the multidimensional zero-one knapsack problem. *European Journal of Operational Research*, 16 :319–326, 1984.

[54] C. Mancel. *modélisation et résolution de problèmes d'optimisation combinatoire issus d'applications spatiales.* PhD thesis, l'Institut National des Sciences Appliquées, Toulouse, 2004.

[55] S. Martello and P. Toth. Algorithms for knapsack problems. *Annals of Discrete Mathematics*, 31 :70–79, 1987.

[56] S. Martello and P. Toth. A new algorithm for the 0-1 knapsack problem. *Management Science*, 34 :633–644, 1988.

[57] A. Mehrotra, M. A. Trick. A column generation approach for graph coloring. *Informs Journal on Computing*, 8 :344–354, 1996.

[58] M. Minoux. *Programmation mathématique, théorie et algorithmes,* tome 1. Dunod, Paris, 1983.

[59] M. Minoux. *Programmation mathématique, théorie et algorithmes,* tome 2. Dunod, Paris, 1983.

[60] M. Moser, D. P. Jokanović and N. Shiratori. An algorithm for the multidimensional multiple-choice knapsack problem. *IEICE Transactions on Fundamentals of Electronics*, 80(3) :582–589, 1997.

[61] R. Nauss. The 0-1 knapsack problems with multiple choice constraint. *European Journal of Operational Research*, 2 :125–131, 1978.

[62] C. Oliva, P. Michelon and C. Artigues. Constraint and linear programming : Using reduced costs for solving the zero/one multiple knapsack problem. *In International Conference on Constraint Programming, Workshop on Cooperative Solvers in Constraint Programming (CoSolv 01), Paphos, Cyprus*, pages 87–98, 2001.

[63] M. W. Padberg. On the facial structure of set packing polyedra. *Mathematical Programming*, 5 :199–215, 1973.

[64] M. W. Padberg, G. Rinaldi. Optimization of a 532 city symmetric traveling salesman problemby branch-and-cut. *Operations Research Letters*, 6 :1–7, 1987.

[65] M. W. Padberg, G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33 :60–100, 1991.

[66] D. Pisinger. Solving hard knapsack problems. *DIKU, University of Copenhagen, Denmark*, Technical Report 94/24, 1994.

[67] D. Pisinger. An exact algorithm for large multiple knapsack problems. *Journal of Operational Research*, 114 :528–541, 1999.

[68] G. Plateau et M. Elkihel. A hybrid method for the 0-1 knapsack problem. *Methods of Operations Research*, 49 :277–293, 1985.

[69] C. C. Ribeiro and F. Soumis. A column generation approach to the multiple-depot vehicle scheduling problem. *Operations Research*, 42(1) :41–52, 1994.

[70] T. Roy and L. Wolsey. Solving mixed integer programming problems using automoatic reformulation. *Operations Research*, 35 :45–57, 1987.

[71] M. SAVELSBERGH. A branch-and-price algorithm for the generalized assignment problem. *Operations Research*, 45(6) :831–841, 1997.

[72] A. Sbihi. *Les méthodes hybrides en optimisation combinatoire : algorithmes exacts et heuristiques*. PhD thesis, CERMSEM, Université Paris 1, 2003.

[73] A. Sbihi. A best first search exact algorithm for the multiple-choice multidimensional knapsack problem. *Journal of Combinatorial Optimization*, 13 :337–351, 2007.

[74] A. Schrijver. On cutting planes. *Annals of Discrete Mathematics*, 9 :291–296, 1980.

[75] W. Shih. A branch-and-bound method for the multiconstraint knapsack problem. *Journal of the Operational Research Society*, 30 :369–378, 1978.

[76] S.Martello and P. Toth. *Knapsack problems : Algorithms and computer implementations*. Wiley, Chichester, England, 1990.

[77] J. Teghem. *Programmation linéaire*. Ellipses, Bruxelles, 1996.

[78] Y. Toyoda. A simplified algorithm for obtaining approximate solution to zero-one programming problems. *Management Science*, 21 :1417–1427, 1975.

[79] J. M. Van Den Akker, C. A. J. Hurkens and M. W. P. Savelsbergh. Time-indexed formulations for machine scheduling problems : column generation. *Informs Journal on Computing*, 12(2) :111–125, 2000.

[80] F. Vanderbeck. On dantzig-wolfe decomposition in integer programming and ways to perform branching in a branch-and-price algorithm. *Operations Research*, 48(1) :111–128, 2000.

[81] F. Vanderbeck and L. A. Wolsey. An exact algorithm for IP column generation. *Operations Research letters*, 19 :151–159, 1996.

[82] L. A. Wolsey. Facets for linear inequalities in 0-1 variables. *Mathematical Programming*, 8 :165–178, 1975.

[83] L. A. Wolsey. Facets for strong valid inequalities for integer programs. *Operations research*, 24 :367–372, 1976.

[84] E. Zemel. Lifting the facets of zero-one polytopes. *Mathematical Programming*, 15 :268–277, 1978.

Méthodes de résolution hybrides pour les problème de type knapsack

Résumé

Dans cette thèse, nous nous intéressons aux problèmes du knapsack multidimensionnel à choix multiple. Ils interviennent essentiellement en télécommunication. Nous proposons de nouvelles méthodes hybrides de résolution exacte et approchée.

Dans un premier temps, nous proposons des méthodes heuristiques en se basant sur les techniques de génération de colonnes et d'arrondi. Ensuite, nous abordons une méthode de recherche locale, dite méthode de branchement local, où des contraintes linéaires sont introduites pour intensifier et diversifier la recherche. Cette méthode est ensuite hybridée avec la génération de colonnes et une technique d'arrondi.

Concernant la résolution exacte, nous nous basons sur une méthode de "Branch and cut". Nous commençons par proposer de nouvelles contraintes valides pour le problème. Ensuite, nous les associons à des contraintes de couverture locales et globales dans un schéma énumératif.

Les approches heuristiques et l'algorithme exact que nous proposons sont comparés à d'autres heuristiques de la littérature et au Solveur de programmes linéaires Cplex . L'ensemble de ces tests numériques ont été menés sur des instances ardues de la littérature ainsi que sur des instances générées aléatoirement de taille modérée.

Mots-clés : Optimisation combinatoire, Knapsack, Génération de colonnes, Méthodes hybrides, Branchement locales, Branch and cut.

Hybrid methods for knapsack problems

Abstract

In this PhD thesis, we deal with multiple-choice multidimensional knapsack problems. They arise mainly in telecommunication field. We propose new hybrid approach and exact methods to solve them.

In a first time, we propose heuristic methods that are based on column generation and rounding procedures. Then, we address a local search method, denoted by local branching method, were some linear constraints are introduced in order to intensify and diversify the search. This method is then combined with a column generation and rounding procedures.

As for the exact resolution, we focus on a branch and cut method. First, we derive new valid inequalities for the problem. Then, we associate them to local and global cover inequalities in an enumerative scheme.

The proposed methods are compared to the results provided by other algorithms of the literature and to the Cplex Solver. They are analyzed computationally on a set of hard instances of the literature and on a group of random generated instances with moderate size.

Key-words : Combinatorial optimization, Knapsack, Column generation, Hybrid methods, Local branching, Branch and cut.

Université Paris I-Panthéon Sorbonne. Maison des Sciences Economiques MSE, 106-112 Boulevard de l'Hôpital 75647 Paris Cedex 13