Les Méthodes Hybrides en Optimisation Combinatoire :Algorithmes Exacts et Heuristiques

Abdelkader Sbihi

▶ To cite this version:

Abdelkader Sbihi. Les Méthodes Hybrides en Optimisation Combinatoire :Algorithmes Exacts et Heuristiques. Mathématiques [math]. Université Panthéon-Sorbonne - Paris I, 2003. Français. NNT : . tel-00012188

HAL Id: tel-00012188
https://theses.hal.science/tel-00012188

Submitted on 28 Apr 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ PARIS-I PANTHÉON-SORBONNE

THÈSE

(Arrêté du 30 mars 1992)

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ PARIS-I

Spécialité : Informatique

présentée et soutenue publiquement

par

ABDELKADER SBIHI

le 18 décembre 2003

Titre

LES MÉTHODES HYBRIDES EN OPTIMISATION COMBINATOIRE :

ALGORITHMES EXACTS ET HEURISTIQUES

Jury

Pr. Marc BUI, Professeur à l'Université de Paris-VIII	*Rapporteur*
Pr. Van-Dat CUNG, Professeur à l'ENSGI-INPG, Grenoble	*Rapporteur*
Pr. Mhand HIFI, Professeur à l'Université d'Amiens	*Directeur*
Pr. Chu-Min LI, Professeur à INSSET, Université d'Amiens	*Examinateur*
Pr. Bernard MONJARDET, Professeur à l'Université de Paris-I	*Président*
Dr. Slim SADFI, Ingénieur, Chercheur associé au CERMSEM	*Examinateur*

On a souvent plus d'envie de passer pour officieux que de réussir dans les offices, et souvent on aime mieux pouvoir dire à ses amis qu'on a bien fait pour eux que de bien faire en effet.

Madame de Sablé.

Cette thèse de doctorat est particulièrement dédiée :

À mes chers parents,

À ma bien aimée femme,

À mes chers frères,

À tous ceux qui ont contribué de près ou de loin à la réalisation et à l'aboutissement de ce travail.

Remerciements

Même s'il est d'usage de commencer par remercier son directeur de thèse, mes remerciements pour M. Mhand Hifi Professeur à l'université de Picardie-Jules Verne vont plus loin que les platitudes habituelles pour avoir accepté de diriger cette thèse. Je ne saurais être suffisamment reconnaissant pour le soutien constant tout au long de ces années que j'ai passées sous sa direction dans la préparation de cette thèse. Aussi bien sa disponibilité que son degré d'exigence ont été pour beaucoup dans l'achèvement de ce travail de recherche.

Je tiens à remercier tout particulièrement M. Bernard Monjardet Professeur émérite à l'université de Paris-I Panthéon-Sorbonne pour avoir accepté de présider ce jury et pour l'intérêt qu'il porte à ce travail.

Que M. Van-Dat Cung Professeur à I'ENSGI-INPG de Grenoble, trouve ici l'expression de ma reconnaissance pour l'intérêt qu'il a porté à mon travail en acceptant de participer au jury et d'être l'un de ses rapporteurs.

C'est avec grand plaisir que j'exprime mes remerciements à M. Marc Bui Professeur à l'université de Paris-VIII Saint-Denis pour sa participation à ce jury et d'être rapporteur des travaux présentés.

Je remercie M. Chu-Min Li Professeur à l'INSEET, universite de Picardie-Jules Verne pour l'intérêt qu'il porte à mes travaux et pour avoir accepté de faire partie de ce jury en tant qu'examinateur.

Je suis très sensible à la présence dans ce jury de M. Slim Sadfi, ingénieur et chercheur, avec qui j'ai lié des liens d'amitié et de travail et pour son soutien moral.

Je tiens aussi à remercier tous les membres du laboratoire CERMSEM UMR-CNRS 8095 permanents ou thésards (et ex-), avec qui j'ai passé des années sympathiques. Qu'il me soit ici pardonné de ne pas les avoir tous nominativement cités.

Une pensée à Marie-Lou et Tuyen pour le formidable travail qu'elles accomplissent au sein du laboratoire CERMSEM.

Une pensée des plus amicales à la famille Faron (Joél et Marie-Laure).

Enfin, je ne saurais être assez reconnaissant pour ma bien aimée femme Aziza pour sa patience et sa disponibilité dans mes moments de doûte et pour le soutien qu'elle a su me donner et dont j'ai bénéficié tout au long de ces moments de préparation.

Mes pensées et pas des moindres vont à mes chers parents pour leur éducation et leurs permanents encouragements et à mes chers frères et tout particulièrement à mon frère Younès qui a tenu à êre présent le jour de la soutenance de cette thèse de doctorat.

Table des matières

1 Introduction **1**

2 Les problèmes de type sac-à-dos **5**

2.1 Introduction 6

2.2 Le problème de sac-à-dos 6

2.2.1 Les méthodes heuristiques 7

2.2.2 Calcul de bornes supérieures et élément critique 8

2.2.3 Les méthodes de résolution exacte 9

2.3 Le problème de la distribution équitable 12

2.4 Le problème de sac-à-dos généralisé à choix multiple 16

2.4.1 Présentation et formulation mathématique 16

2.4.2 Une méthode de séparation et évaluation pour le MMKP 17

2.4.3 Méthodes de résolution approchée pour le MMKP 21

3 Un algorithme approché pour le problème de la distribution équitable **27**

3.1 Introduction 27

3.2 Une première méthode approchée 29

3.2.1 Les éléments critiques et le problème de la distribution équitable 29

3.2.2 Représentation de la solution 30

3.2.3 Construction d'une solution initiale 32

3.2.4 L'algorithme 33

3.3 Une deuxième méthode approchée 38

3.3.1 Introduction d'une deuxième profondeur 38

3.3.2 Le principe de l'algorithme 40

3.4 Partie expérimentale 41
3.4.1 Sommaire des résultats 43
3.4.2 Discussion des tests numériques 45
3.5 Conclusion 49

4 Méthodes de recherche locale pour le sac-à-dos généralisé à choix multiple 51

4.1 Introduction 51
4.2 Méthodes de recherche locale 52
4.3 Représentation de la solution 53
4.3.1 Une solution initiale 54
4.3.2 Discussion de la non réalisablité d'une solution 56
4.3.3 Une recherche locale complémentaire 57
4.4 La recherche locale guidée 61
4.5 Adaptation de la recherche locale guidée 62
4.6 Un algorithme utilisant la pénalisation et la transformation normalisante 63
4.6.1 Le principe général de l'algorithme 63
4.6.2 Description de l'algorithme DerAlg 64
4.7 Partie expérimentale 66
4.7.1 Performances des procédures CP et CCP 67
4.7.2 Performance de l'algorithme DerAlg 70
4.8 Une recherche locale réactive 77
4.8.1 Un algorithme de recherche locale réactive 77
4.8.2 Représentation d'une solution 78
4.8.3 Principe de la recherche locale adaptée 78
4.8.4 Description de l'algorithme 79
4.9 Un algorithme modifié de recherche locale et réactive 80
4.10 Partie expérimentale 82
4.10.1 Comportement des deux versions de l'approche 82
4.10.2 Performance des approches RLS et MRLS 85
4.11 Conclusion 90

5 Une méthode exacte pour le problème de sac-à-dos généralisé à choix multiple 91

5.1 Introduction 91
5.2 Les bornes supérieures et inférieures 92
5.3 Algorithme de séparation et d'évaluation 96
5.3.1 Développement de l'arborescence de recherche 96
5.3.2 Principe de l'algorithme 98
5.3.3 Troncature des branches de l'arborescence 99
5.3.4 L'algorithme 101
5.4 Partie expérimentale 101
5.4.1 Génération des instances 102
5.4.2 Performance de l'algorithme exact 104
5.5 Conclusion 105

6 Conclusion et perspectives 107

6.1 Conclusion 107
6.2 Perspectives 109
6.2.1 Fixation des variables pour le sac-à-dos généralisé à choix multiple109
6.2.2 Relaxation lagrangienne et montée surrogate 111
6.2.3 Méthodes évolutionnistes 112

Bibliographie 115

Table des figures

2.1 Heuristique gloutonne pour le KP. 8
2.2 Algorithme de séparation et évaluation en profondeur pour le KP. . . . 11
2.3 Heuristique de Yamada et Futakawa. 15
2.4 Représentation binaire d'une solution de MMKP. 19
2.5 Algorithme de résolution par séparation et évaluation pour le MMKP. . 20
2.6 Procédure Heu1 : une première heuristique pour le MMKP. 22
2.7 Procédure AlgApp : une deuxième heuristique pour le MMKP. 24

3.1 Représentation d'une solution du KSP. 30
3.2 La procédure GH : construction d'une (première) solution réalisable. . . 32
3.3 Représentation d'une solution partielle réalisable du KSP pour la classe J_i. Le symbole $*$ représente les éléments libres non encore fixés pour une classe donnée. 34
3.4 La première version de l'algorithme : utilisation d'une profondeur. . . . 36
3.5 Représentation d'une solution partielle de KSP utilisant deux profondeurs pour une classe J_i. 39

4.1 Représentation d'une solution de MMKP. 54
4.2 La procédure constructive : CP. 56
4.3 La procédure complémentaire CCP. 59
4.4 L'approche GLS. 62
4.5 Un algorithme avec pénalisation et normalisation : DerAlg 65
4.6 Variation du pourcentage d'amélioration de DerAlg versus CCP. 74
4.7 Variation du pourcentage de déviation de KLMA et DerAlg. 75

4.8 Représentation d'une solution du MMKP. 78
4.9 L'algorithme RLS pour le MMKP. 79
4.10 Modification de l'algorithme RLS en ajoutant une liste mémoire : MRLS. 81
4.11 Comparaison des temps d'exécution consommés par les approches RLS et MRLS. 89
4.12 Comparaison des pourcentages d'amélioration de RLS et MRLS. 89

5.1 Schéma de développement de l'arborescence. 97
5.2 Schéma de développement d'un nœud. 98
5.3 Premier schéma de troncature d'un nœud de l'arborescence. 100
5.4 Deuxième schéma de troncature d'un nœud de l'arborescence. 100
5.5 Algorithme de résolution par séparation et évaluation pour le MMKP. . 101

Liste des tableaux

3.1 Détail des instances de test : 1 ≤x≤ 4. 42
3.2 Représentation des résultats obtenus par les trois versions de l'algorithme. 44
3.3 Performance de l'approche recherche à profondeur multiple (MTS) sur les instances non corrélées avec $m \leq 10$ (classes). 46
3.4 Performance de l'approche recherche à profondeur multiple (MTS) sur les instances non corrélées avec $m \geq 20$ (classes). 47
3.5 Performance de l'approche recherche à profondeur multiple (MTS) sur les instances corrélées. 48

4.1 Détails des instances 67
4.2 Performance des approches de Khan *et al.*'s [27] et Moser *et al.*'s [40] sur les instances de la littérature. Le symbole $*$ signifie que la solution optimale n'est pas connue. 68
4.3 Performance des approches CP et CCP. Le symbole $*$ signifie que la solution optimale n'est pas connue. Le symbole $<$ signifie que le temps d'exécution T est négligeable. 69
4.4 Le comportement de DerAlg par rapport à la variation du paramètre $MaxIter$. 71
4.5 Effets du paramètre D sur l'algorithme DerAlg. 72
4.6 Le comportement de DerAlg quand on fait varier π. 72
4.7 Performance de DerAlg testé sur les instances du Tableau 4.1 et comparé aux résultats de KLMA. Le symbole $\circ$ signifie que la solution optimale (ou la meilleure) est atteinte et le symbole $\triangleright$ signifie que DerAlg améliore la solution produite par KLMA. 73

4.8 Comportement de RLS selon la variation du nombre d'itérations $MaxIter$ et en fixant p à 5. 84
4.9 Comportement de RLS selon la variation de p. 84
4.10 Comportement de MRLS selon la variation dynamique de la longueur de la liste mémoire. 85
4.11 Sommaire des résultats obtenus par DerAlg, RLS et MRLS. Le symbole $*$ signifie que la solution optimale n'est pas connue. 86
4.12 Résultats numériques obtenus par RLS et MRLS. Le symbole $*$ signifie que la solution optimale n'est pas connue. 87

5.1 Résultats numériques du premier groupe 103
5.2 Résultats numérique du second groupe 103
5.3 Résultats numérique du troisième groupe 103
5.4 Résultats numériques du quatrième groupe 105

Chapitre 1

Introduction

Imagination is more important than knowledge.

Albert Einstein

Dans la vie quotidienne on est souvent confronté à toutes sortes de problèmes d'ordre économique, industrielle, militaire, etc. Un problème peut être décrit et représenté sous forme d'un langage formel. Par exemple, plusieurs problèmes peuvent être formulés sous forme d'un problème d'optimisation combinatoire : il s'agit, en général, de maximiser (problème de maximisation) ou de minimiser (problème deminimisation) une fonction objectif sous certaines contraintes. Proposer des méthodes (ou approches) de résolution pour ce type de problèmes revient à considérer deux points majeurs : la *qualité de la solution* et le *temps d'exécution*. En général, la qualité de la solution est elle aussi fonction du temps. Généralement, les méthodes que l'on met en œuvre pour résoudre un problème d'optimisation combinatoire dépendent de la complexité de ce dernier. La théorie de la NP-complétude (Garey et Johnson[11]) fournit de précieux renseignements sur le genre de méthodes à adopter en fonction de la difficulté intrinsèque des problèmes. Plusieurs choix sont possibles pour traiter certains problèmes de l'optimisation combinatoire. Nous retiendrons les choix suivants (parmi ceux que recensent Barthélemy, Cohen et Lobstein dans[2]) :

– appliquer des algorithmes exacts dont la complexité peut croître exponentiellement avec la taille des instances à traiter ;
– concevoir des heuristiques, c'est-à-dire des algorithmes qui tentent de fournir une solution approchée, souvent assez bonne mais pas nécessairement optimale, en

un temps de calcul “raisonnable”.

Ce sont ces types d’approches que nous avons été amenés à suivre pour la résolution d’une classe de problèmes de l’optimisation combinatoire : les problèmes du type sac-à-dos. En effet, il s’agit d’une classe de problèmes où chaque problème est formulé comme un programme linéaire en nombres entiers. Un programme linéaire en nombres entiers est un problème dans lequel certaines ou toutes les variables doivent être entières et positives ou nulles. Tout programme linéaire en nombres entiers ne comprenant qu’une seule contrainte fonctionnelle est appelé un problème de sac-à-dos ou “knapsack” (ou “knapsack unidimensionnel”). A cause de son utilité, ce problème particulier est de plus en plus utilisé dans le domaine décisionnel. Le domaine d’application de ce problème (ainsi que ses variantes) inclut des cas du domaine de transport, de la logistique, de la fiabilité ainsi que de la production. Le problème du sac-à-dos est parfois considéré comme un sous problème ou une relaxation de problèmes plus complexes. Cette thèse comporte deux parties. La première partie décrit l’étude de deux variantes du problème de sac-à-dos en se basant sur les méthodes de résolution approchée. La deuxième partie entreprend l’étude d’une variante du problème de sac-à-dos en s’appuyant sur une procédure de séparation et évaluation (“branch and bound”). Plus précisément, la première partie comporte deux chapitres. Le premier chapitre (chapitre 3) présente une méthode approchée pour le problème de la distribution équitable (“knapsack sharing”). Dans un premier temps, nous proposons un algorithme exploitant certaines caractéristiques de la recherche tabou. Cet algorithme est considéré comme une version simple dans lequel nous utilisons (i) un paramètre profondeur limitant la recherche des solutions voisines autour des éléments dits “critiques” et (ii) une mémoire (liste tabou) permettant la non stagnation de certaines solutions. Dans un deuxième temps, nous proposons une version améliorée de l’algorithme. Nous introduisons cette fois-ci une stratégie combinant l’intensification de la recherche dans l’espace des solutions et la diversification de la solution. Il s’agit principalement d’introduire une modification dans la façon d’appliquer le paramètre profondeur utilisé auparavant dans la version simple de l’algorithme. Le deuxième chapitre (chapitre 4) présente l’étude du problème de sac-à-dos généralisé à choix multiple ou encore le problème du sac-à-dos multicontraintes à choix multiple (“multiple-choice multidimesional knapsack”). Dans ce chapitre, nous proposons deux méthodes de résolution. Une première méthode qui s’appuie sur une “recherche guidée” qui est par la suite combinée avec une stratégie

de pénalisation sur des composantes de la fonction objectif. Une deuxième méthode qui s'appuie principalement sur la recherche locale. La recherche locale appliquée au problème peut être vue comme une approche en deux phases : (i) une première phase qui consiste à faire des inter-changes pour tenter d'améliorer la solution en cours et (ii) une deuxième phase qui consiste à faire une diversification de la solution en acceptant une éventuelle dégradation de la solution en cours.
En effet, pour la première méthode (la première partie du chapitre4), nous proposons une procédure constructive qui tente de produire une solution réalisable pour le problème. Dans le cas où la solution obtenue est réalisable, nous appliquons une procédure s'appuyant sur la stratégie d'inter-change afin d'améliorer la qualité de la solution initiale. L'application de cette dernière sur une solution non réalisable permet aussi de réduire la non-réalisabilité de la solution. Dans un deuxième temps, nous proposons un algorithme de pénalisation qui s'appuie principalement sur une recherche guidée. Cet algorithme est composé de deux phases principales : (i) une première phase qui consiste à perturber le problème original (en pénalisant certaines composantes de la fonction objectif) et (ii) une deuxième phase qui applique une recherche locale sur la solution perturbée en cours afin de tenter de produire une nouvelle solution de meilleure qualité. A la fin de cette phase, une transformation simple permet de récupérer une solution pour le problème original. Ce procédé (les deux phases) est réitéré un certain nombre de fois afin d'améliorer la solution. Dans la deuxième partie du chapitre 4, nous proposons un algorithme de résolution approchée en se basant sur différentes phases. Une première phase qui consiste à partir d'une solution initiale et d'effectuer une recherche locale utilisant la notion d'inter-change. Une deuxième phase est appliquée sur chaque solution estimée non-améliorante sur le voisinage en cours ; dans ce cas, nous utilisons une procédure débloquante afin de tenter d'esquiver la solution en cours. Une troisième phase consiste à effectuer une diversification en utilisant une procédure permettant de dégrader la solution en cours. Finalement, dans une dernière phase, nous introduisons une mémoire (liste tabou) afin d'éviter les recyclages sur certaines solutions explorées auparavant. Ces différentes étapes sont répétées un certain nombre de fois pour tenter de produire une solution de “bonne qualité”.
La deuxième partie de cette thèse est présentée sous forme d'un chapitre (chapitre 5) qui entreprend l'étude du problème de sac-à-dos généralisé (multidimensionnel) à choix multiple. Nous proposons un algorithme de résolution exacte en s'appuyant sur une

procédure de séparation et évaluation ("branch-and-bound"). Dans un premier temps, nous commençons l'étude par la réduction du problème original au problème de sac-à-dos à choix multiple qui sera considéré par la suite comme un problème auxiliaire pour le problème initial. Dans la même partie, nous présentons une borne supérieure ainsi qu'une borne inférieure de départ pour le problème. Dans un deuxième temps, nous présentons une procédure de séparation et évaluation en s'appuyant sur une recherche par le meilleur d'abord. Finalement, nous présentons une étude expérimentale en montrant l'efficacité de la méthode sur différents groupes d'instances de petite et moyenne taille.

Finalement, le chapitre 6 résume les travaux effectués au cours de cette thèse et propose une ouverture sur de nouvelles directions de recherche à court terme ainsi qu'à long terme.

Chapitre 2

Les problèmes de type sac-à-dos

Le problème de sac-à-dos (ou knapsack problem) est un problème classique d'optimisation appartenant à la classe des problèmes NP-difficiles. On retrouve ce problème sous de nombreuses formes, comme par exemple, comment loger des morceaux de musique dans une cassette de sorte à avoir la plus longue durée de musique.
Le problème est de remplir un sac-à-dos supportant un poids maximum c avec des objets ayant un poids w_j, pour $j = 1, \ldots, n$, et un indice de satisfaction de telle sorte que la satisfaction totale (fonction objectif) $Z(x)$ soit maximale. La question qui se pose est de savoir quels objets doit-on mettre dans le sac ?
Ce problème intervient comme un sous-problème dans plusieurs problèmes, par exemple, les problèmes de découpe (voir Gilomre et Gomory [12]) lors de la génération d'un pivot du simplexe.
Sous le terme de "sac-à-dos" sont regroupées différentes variantes qui, 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 les diverses méthodes de résolution du sac-à-dos unidimensionnel et nous présentons ses variantes que nous avons étudiées ainsi que l'essentiel des approches de résolution existantes.

2.1 Introduction

Dans la littérature, il existe plusieurs versions du problème du sac-à-dos. On privilégiera dans cette thèse, la formulation en $0-1$. On considère un ensemble d'objets étiquetés de 1 à n. Chaque objet $j \in \{1, \ldots, n\}$ dispose d'un poids w_j de valeur entière et d'un profit v_j (réel *a priori*). On dispose d'un sac-à-dos dont le contenu ne peut excéder une capacité c entière. On désire le remplir de façon à maximiser la somme des utilités des objets emportés, en respectant la contrainte de capacité. Autrement dit, on désire résoudre le problème de programmation linéaire entière suivant :

$$x \in arg \max_{j\in\{1,\ldots,n\}} \Big\{ \sum_{1\leq j\leq n} v_j x_j, \text{ sous la contrainte} \sum_{1\leq j\leq n} w_j x_j \leq c \Big\}.$$

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 objective :

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

Nous formulons l'hypothèse suivante :

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

2.2 Le problème de sac-à-dos

Dans ce paragraphe, on rappelle le problème de sac-à-dos, KP (Knapsack Problem), tel qu'il a été défini dans l'introduction. Il servira de problème générique tout au long de l'étude rapportée dans cette thèse.
On appelle une instance d'un problème de sac-à-dos, la donnée des n profits v_j pour $j = \ 1, \ldots, n$, des n poids w_j pour $j = 1, \ldots, n$ et de la capacité c. Tout vecteur binaire $(x_1, x_2, \ldots, x_n)$ est appelé solution du problème du sac-à-dos, KP. On dit qu'une solution est réalisable, notée $\bar{x}$, si elle vérifie la contrainte de capacité, c'est-à-dire $\sum_{1\leq j\leq n} w_j \bar{x}_j \leq c$. La solution est dite optimale, notée $x^\star$, 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 :

$$\sum_{j=1}^{n} v_j \bar{x}_j \leq \sum_{j=1}^{n} v_j {x^\star}_j.$$

La formulation du problème du sac-à-dos unidimensionnel, KP, en variables binaires est la suivante :

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

Diverses approches aussi bien exactes qu'approchées ont été développées pour résoudre ce problème (voir Fayard et Plateau [9], Martello et Toth [38], [37] et Pisinger [43]).

2.2.1 Les méthodes heuristiques

Parmi les méthodes heuristiques, on peut citer la méthode dite gloutonne. Un algorithme glouton construit une solution de manière incrémentale, en faisant à chaque pas un choix maximisant (si l'objectif est de maximiser) une fonction objectif.

Comme il existe plusieurs variantes pour ces méthodes gloutonnes, nous présentons ici une version simple et simplifiée. Tout d'abord, on ordonne les objets selon l'ordre décroissant du rapport profit par poids, c'est-à-dire :

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \geq \frac{v_j}{w_j} \geq \ldots \geq \frac{v_n}{w_n}.$$

L'algorithme glouton que nous présentons ici consiste donc à sélectionner à chaque étape un élément selon l'ordre précédemment défini. Si l'élément est admissible, c'est-à-dire si son poids ne dépasse pas la capacité restante (résiduelle) aprés fixation des autres éléments, alors, il est mis dans le sac sinon, on sélectionne l'élément qui se situe juste après et qui peut être admissible et ainsi de suite de proche en proche jusqu'à épuisement de tous les objets pouvant être mis dans le sac.

Ci-après, l'heuristique gloutonne. Notons que cette méthode n'est pas la seule méthode gloutonne de résolution pour le problème du KP.

Entrée : Une instance d'un problème de sac-à-dos ;
Sortie : Une solution réalisable $\bar{x}$;
1. $\bar{c} := c$;
2. Pour $j := 1$ jusqu'à n faire
Si $w_j \leq \bar{c}$ alors $\bar{x}_j := 1$; $\bar{c} := \bar{c} - w_j$;
sinon $\bar{x}_j := 0$.
FinSi
FinPour
3. Sortir avec une solution réalisable $\bar{x}$

FIG. 2.1 – Heuristique gloutonne pour le KP.

2.2.2 Calcul de bornes supérieures et élément critique

Calculer des bornes supérieures ou inférieures (en fonction du contexte d'optimisation) permet d'encadrer la valeur de la solution optimale pour les problèmes que l'on tente de résoudre. Ensuite, elles sont utilisées pour le développement de méthodes de résolution exacte s'appuyant sur des procédures d'énumération implicite (ou méthodes de séparation et évaluation).

Dantzig [7] a proposé de calculer une borne supérieure pour le KP. Pour ce faire, l'idée consiste à relâcher chaque variable x_j pour $j = 1, \ldots, n$ du problème KP dans l'intervalle $[0, 1]$: le problème relâché est donné par :

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

La résolution du problème LP(KP) consiste à remplir le sac-à-dos objet après objet et de proche en proche jusqu'à sa saturation. Ensuite, on repère le premier objet ne

pouvant être mis en totalité dans le sac. On appelle cet objet l'*élément critique* x_ℓ. Il s'agit d'un élément d'indice $\ell \in \{1, \ldots, n\}$ tel que $\sum_{j=1}^{\ell-1} w_j \leq c < \sum_{j=1}^{\ell} w_j$.

La solution de LP(KP) se présente alors comme suit :

$$\bar{x}_j := \begin{cases} 1 & \text{si } j = 1, \ldots, \ell - 1, \\ \frac{(c - \sum_{j=1}^{\ell-1} w_j)}{w_\ell} & \text{si } j = \ell, \\ 0 & \text{si } j = \ell + 1, \ldots, n. \end{cases}$$

La borne supérieure pour le problème du KP notée par UB_d, est la solution optimale de LP(KP). Une solution réalisable pour KP l'est aussi pour LP(KP). Ceci est vrai puisque la contrainte d'intégralité sur les variables x_j pour $j = 1, \ldots, n$ de LP(KP) est plus relâchée que celle de KP. On a :

$$Z_{LP(KP)}(x) = UB_d \geq Z_{KP}(x).$$

Sachant que toute solution de KP est entière, alors la valeur entière inférieure de la solution optimale de LP(KP) demeure une borne supérieure pour le KP. Nous avons donc :

$$UB_d = \sum_{j=1}^{\ell-1} v_j + \left\lfloor \frac{(c - \sum_{j=1}^{\ell-1} w_j)}{w_\ell} v_\ell \right\rfloor. \tag{2.1}$$

Cette borne est communément appelée la borne de Dantzig [7].
Il peut y exister d'autres améliorations de cette borne. Parmi les plus connues est la borne proposée par Martello et Toth [36].

2.2.3 Les méthodes de résolution exacte

Plusieurs approches de résolution exacte pour le problème du KP ont été élaborées. La plupart de ces méthodes sont basées sur les techniques énumératives. En général, on s'appuie sur une méthode de séparation et évaluation. Il s'agit d'énumérer d'une manière implicite les solutions et d'en choisir la meilleure parmi toutes.

La méthode de séparation et évaluation

Le concept général de ces méthodes se base sur une structure arborescente de recherche de solutions. Chaque nœud de l'arborescence sépare l'espace de recherche en deux sous-espaces et de proche en proche, jusqu'à l'exploration totale de l'espace des solutions.
Développer tout l'espace de recherche, consiste pour une instance de n éléments, à développer 2^n vecteurs binaires de taille n. Ce qui est excessivement lourd et irréalisable au regard du temps d'exécution nécessaire à l'exploration de toute l'arborescence.
Les méthodes de séparation et évaluation se basent, en général, sur les principes suivants mais diffèrent entre elles par le choix de stratégies à opérer pour la séparation, le développement de l'arborescence et l'utilisation de la fonction d'évaluation :

1. **Séparation :** ce principe s'effectue au niveau de l'instance du sac-à-dos. L'élément de séparation peut être pris selon un ordre prédéfini ou selon un élément défini pendant l'exploration. Le choix de l'élément de séparation est important pour améliorer les performances de l'algorithme.
2. **Développement :** ce principe se fait selon une stratégie de développement qui peut être en profondeur ou par le meilleur d'abord.
3. **Fonction d'évaluation :** son choix est primordial puisqu'elle permet de réduire l'espace de recherche. Selon les cas, il est parfois préférable de choisir une fonction d'évaluation facile à calculer qu'une autre qui peut donner de meilleurs résultats mais qui soit moins facile à calculer. L'idéal est de trouver un meilleur compromis entre la qualité et la difficulté de calcul d'une fonction d'évaluation.

Dans ce qui suit nous présentons l'algorithme de séparation et évaluation développé par Horowitz et Sahni [26]. Cette méthode 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 $x_j = 0$ correspond à l'élément j qui n'est pas mis dans le sac.

Le développement de l'arborescence 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 UB_d présentée dans la section 2.2.2. 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 2.2 représente les principales étapes de cet algorithme.

Entrée : Une instance d'un problème de sac-à-dos ;
Sortie : Une solution optimale $x^\star$;

1. $j := 1$; *MeilleureValeur* :=0 ;
2. Calcul de la borne supérieure UB_d qu'on peut atteindre depuis j
3. **Si** $UB_d \geq MeilleureValeur$ **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. $j := max\{\ell : \ \ell \leq j$ et $\underline{x}'_\ell = 1\}$
   Tant que j existe **faire**.
   $\underline{x}'_j := 0$;
   Fin Tant Que ;
5. $x^\star := \underline{x}$.

FIG. 2.2 – Algorithme de séparation et évaluation en profondeur pour le KP.

2.3 Le problème de la distribution équitable

Dans cette partie, nous présentons la modélisation du problème de la distribution équitable (le Knapsack Sharing Problem : KSP). Il s'agit d'un problème en $max - min$. Ce problème possède un bon nombre d'applications dans les domaines du commerce, du transport, des communications, d'allocation des ressources, etc. (voir Brown [3, 4] et Tang [55]). Pour ce problème, la distribution équitable ne consiste pas à partager en parts égales les ressources mais plutôt à maximiser la plus petite part. Autrement dit privilégier le moins favorisé.

Prenons l'exemple de l'État qui a le souci de partager équitablement le budget c alloué aux différents ministères qui sont d'un nombre fixé à m. Chaque ministère possède ses différentes priorités pour la réalisation d'un certain nombre de projets. Allouer équitablement le budget aux différents ministères, ne revient pas à diviser le budget c en parts égales, i.e. $(\frac{c}{m})$, entre les différents départements ministériels. Mais comme chacun des projets d'un ministère i, $i = 1, \ldots, m$, dispose d'un coût de réalisation et d'un profit estimé, alors le problème de partage équitable revient à maximiser la plus petite valeur du profit global par ministère sans dépasser le budget prévu pour la réalisation de ces projets (d'autres exemples sont présentés dans Brown [3] et Tang [55]).

Plus précisément, nous considérons dans cette partie le problème KSP à variables binaires. Pour ce problème, à chaque objet j sont associés un profit v_j et un poids w_j. On dispose d'un ensemble $\mathcal{N}$ de n objets où $\mathcal{N}$ est composé de m classes d'objets disjointes. Si J_i désigne la i-ème classe d'objets, $i = 1, \ldots, m$, alors $\forall p = 1, \ldots, m, \forall q = 1, \ldots, m$, pour $p \neq q$, $J_p \cap J_q = \emptyset$ et $\cup_{i=1}^{m} J_i = \mathcal{N}$.

L'objectif de ce problème est de déterminer le sous-ensemble d'objets que nous allons retenir dont la capacité a pour valeur c. Ce sous-ensemble est choisi de manière à maximiser la valeur de la fonction objectif qui réalise le minimum par rapport à l'ensemble de toutes les classes.

Soit x_j la variable de décision à valeur binaire, alors $x_j = 1$ si l'objet j est pris dans le sac, et $x_j = 0$ sinon. Alors le problème KSP peut être formulé comme suit :

$$(KSP)\begin{cases} \text{Maximiser} & \min\limits_{1\leq i\leq m} \Big\{\sum\limits_{j\in J_i} v_j x_j\Big\} \\ s.c. & \sum\limits_{j\in\mathcal{N}} w_j x_j \leq c \\ & x_j \in \{0,1\},\ \text{pour } j=1,\ldots,n. \end{cases}$$

Les classes sont indexées de J_1 à J_m et les éléments d'une classe $J_i,\ i = 1,\ldots,m$, sont indexés de 1 à $|J_i|$. Afin d'éliminer les cas triviaux, nous considérons lors de notre étude que $\sum_{j\in\mathcal{N}} w_j > c$.

Le KSP est un problème NP-complet et on le retrouve dans la littérature sous la notation KSP($Bn/m/1$) (voir Yamada et Futakawa [62] et Hifi et Sadfi [22]), ce qui signifie que nous disposons de n objets de type binaire (B), répartis sur m classes et avec une seule contrainte.

Le problème d'allocation max-min à été largement étudié (Brown [4], Kuno *et al.* [30], Luss [31], Pang et Yu [42] et Tang [55]). Différentes approches, exactes et approchées, ont été développées et adaptées spécialement pour ce problème. Pour le KSP en variables continues, Kuno *et al.* [30] ont proposé un algorithme linéaire. Yamada et Futakawa [62] donnent un algorithme différent pour le KSP($Cn/m/1$). Jusqu'à présent, peu d'algorithmes exacts ou approchés ont été proposés pour le KSP à variables binaires. En effet, Yamada et Futakawa [62] ont étendu l'approche développée pour le KSP($Cn/m/1$) au problème KSP($Bn/m/1$). Les résultats expérimentaux montrent que l'approche donne de bons résultats. Pour le même problème, Yamada *et al.* [63] ont proposé deux algorithmes exacts : le premier utilisant la méthode de séparation et évaluation et le deuxième utilisant une adaptation de la méthode de recherche binaire. Les auteurs ont montré que leur deuxième algorithme, basé sur la méthode de recherche binaire, était plus performant comparé avec leur algorithme basé sur la méthode de séparation et évaluation. Finalement, Hifi et Sadfi [22] ont proposé une approche basée sur la méthode de programmation dynamique, dans laquelle le problème de base est décomposé en une série de problèmes de sac-à-dos. Les auteurs ont montré que leur approche donnait de bons résultats dans le sens qu'elle permet de résoudre certaines instances de grande taille.

Dans le chapitre 3 nous nous intéressons principalement à la résolution approchée du KSP. Pour cela, nous présentons dans ce qui suit les étapes principales d'un algorithme

approché proposée par Yamada et Futakawa [62].

Un algorithme approché pour le KSP

Yamada et Futakawa [62] ont proposé une méthode gloutonne, c'est-à-dire que la recherche d'une solution se fait pas à pas. A chaque étape, elle complète une solution réalisable partielle obtenue à l'étape précédente.

L'algorithme s'appuie principalement sur les étapes suivantes.
Soit le problème auxiliaire $Aux_k(c^k)$ suivant :

$$(Aux_k(c^k)) \begin{cases} \text{Maximiser} & \sum_{j \in J_k} v_j x_j \\ s.c. & \sum_{j \in \mathcal{N}} w_j x_j \leq c^k \\ & 0 \leq x_j \leq 1, \text{ pour } j \in J_k. \end{cases}$$

où $Aux_k(c^k)$ est un problème de sac-à-dos relaché et considérons les notations suivantes (qui seront utilisées dans l'algorithme).

Notations

$\bar{x}^k(c^k)$: représente la solution du problème $Aux_k(c^k)$;
$\bar{z}^k(c^k)$: représente l'évaluation de la solution $\bar{x}^k(c^k)$;
n_k : représente la cardinalité de l'ensemble J_k, i.e, $|J_k|$;
$v_{k,j}$: est le profit v_j de l'élément j appartenant à la classe J_k ;
$w_{k,j}$: est le poids v_j de l'élément j appartenant à la classe J_k ;
$x_{k,j}$: dénote la variable associée à l'élément j appartenant à la classe J_k ;
On pose $w_j^k = \sum_{l=1}^{j} w_{k,l}$ et $v_j^k = \sum_{l=1}^{j} v_{k,l}$.

Soit maintenant le problème $\bar{C}(KSP)$ défini comme suit :

$$\bar{C}(KSP) \begin{cases} \text{Maximiser} & \min_{1 \leq k \leq m} \left\{ \sum_{j \in J_k} \bar{z}^k(c^k) \right\} \\ s.c. & \sum_{k=1}^{m} c^k \leq c \\ & c^1, \ldots, c^m \geq 0. \end{cases}$$

Soit $(\bar{c}^1, \ldots, \bar{c}^m)$ une solution optimale de $\bar{C}(KSP)$ de valeur $\bar{z}$, et soient les valeurs u_k, $k = 1, \ldots, m$, définies de la façon suivante :

$$u_k = min\{j : \ v_j^k \geq \bar{z}\}.$$

Dans la figure 2.3, nous décrivons l'algorithme approché de Yamada et Futakawa [62] :

Entrée : Les vecteurs (n_j), $(w_{k,j})$, $(v_{k,j})$, c, (u_j) et $\hat{x}$;
Sortie : La solution $\underline{x}$ et son évaluation $\underline{z}$;

1. $\underline{x} := \hat{x}$;
   $w^k := w^k_{u_k-1}$, $v^k := v^k_{u_k-1}$, $j_k := u_k$ pour $k = 1, \ldots, m$;
2. Répéter
3. Si $w_{(k_{min}, j_{k_{min}})} \leq c - \sum_{k=1}^{m} w^i$ alors
   $\underline{x}_{(k_{min}, j_{k_{min}})} := 1$;
   $v^{k_{min}} := v^{k_{min}} + v_{(k_{min}, j_{k_{min}})}$;
   $w^{k_{min}} := w^{k_{min}} + w_{(k_{min}, j_{k_{min}})}$;
   Sinon $\underline{x}_{(k_{min}, j_{k_{min}})} := 0$;
4. $j_{k_{min}} := j_{k_{min}} + 1$;
   Si $j_k \leq n_k$, $k = 1, \ldots, m$ alors
   $k_{min} := arg \min_{1 \leq k \leq m} \{v_k\}$
5. $\underline{z} := min\{v^1, \ldots, v^m\}$;
6. Jusqu'à $(j_{k_{min}} > n_{k_{min}})$;

FIG. 2.3 – Heuristique de Yamada et Futakawa.

L'heuristique développée utilise les valeurs $u_1, \ldots, u_m$ et considère les éléments triés suivant l'ordre décroissant du rapport profit par poids. En démarrant par la solution réalisable triviale où $\hat{x}_{k,j} = 1$ si $j \leq u_k - 1$, et $x_j = 0$ sinon, pour $k = 1, \ldots, m$, l'heuristique prend les éléments un à un et tente de mettre le maximum d'éléments jusqu'à saturation du sac. A chaque étape, l'élément considéré est le premier élément en respectant l'ordre décroissant des rapports profit par poids des éléments. De plus, cet élément doit appartenir à la classe pour laquelle la valeur de la somme des profits

des éléments déjà mis dans le sac et la plus petite par rapport à la valeur de la somme des profits des éléments des autres classes déjà mis dans le sac.

2.4 Le problème de sac-à-dos généralisé à choix multiple

2.4.1 Présentation et formulation mathématique

Le problème de sac-à-dos généralisé à choix multiple ou problème de sac-à-dos multidimensionnel à choix multiple ou encore problème de sac-à-dos multicontraint à choix multiple est une version bien particulière du problème du KP. Il peut être considéré comme une variante qui généralise encore plus deux problèmes généralisant à leur tour le problème de sac-à-dos : le problème du MDKP [54] et le problème du MCKP [41]. Ces deux généralisations ont bien été étudiées par le passé (voir Martello et Toth [37], Pisinger [43] et Chu et Besaley [6]) pour lesquels diverses approches exactes et heuristiques efficaces on été proposées pour les résoudre.

Nous rappelons que le problème du MMKP est caractérisé par la donnée d'un vecteur capacité ou ressources C=$(C^1, C^2, \ldots, C^m)$, d'un ensemble $\mathcal{J}$ =$\{J_1, \ldots, J_i, \ldots, J_n\}$ d'objets répartis sur n classes disjointes telles que pour chaque couple (p, q), $p \neq q$, $p \leq n$ et $q \leq n$, nous avons $J_p \cap J_q = \emptyset$ et $\cup_{i=1}^{n} J_i = \mathcal{J}$. Chaque classe i, $i = 1, \ldots, n$ [1], est de cardinalité r_i (nombre d'objets appartenant à la dite classe i).
A chaque objet j de la classe i sont associés un profit positif v_{ij} et un vecteur poids $W_{ij} = (w_{ij}^1, w_{ij}^2, \ldots, w_{ij}^m)$. Le but est d'attribuer au sac, exactement *un et un seul* objet par classe dans le but de maximiser la valeur totale du choix sans que l'une ou l'autre des contraintes capacités ne soient à un moment ou un autre violée. Le MMKP peut donc être formulé de la façon suivante :

[1]Nous ne faisons pas de distinction entre une classe J_i donnée et son indice i, sauf quand il y'a confusion dans les notations.

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

La valeur x_{ij} valant 0, signifie que l'objet j de la i-ème classe n'est pas choisi ou valant 1 signifie que l'objet j de la i-ème classe est choisi.
Sans perte de généralité et pour éviter les solutions triviales, nous formulons l'hypothèse suivante :

$$
(H): \quad \sum_{i=1}^{n} \min_{1\leq j\leq r_i}\left\{w_{ij}^{k}\right\} \leq C^k \leq \sum_{i=1}^{n} \max_{1\leq j\leq r_i}\left\{w_{ij}^{k}\right\}, \text{ pour } k=1,\ldots,m.
$$

Nous rappelons que le problème du MMKP est une généralisation des problèmes MDKP et MCKP.
En effet, si $m = 1$, alors le MMKP devient un MCKP et si $n = 1$ et si l'on supprime la contrainte du choix (2), alors le MMKP devient un MDKP. Curieusement, peu de travaux ont été consacrés au problème MMKP (Moser *et al* [40], Khan *et al.* [27] et [28]), contrairement au MDKP (voir Pisinger[43], Chu et Beasley [6]) et MCKP (voir Dudinski [8], Nauss [41] et Pisinger [47]). Dans la suite, nous montrerons la difficulté du problème tant dans sa résolution exacte que dans l'élaboration de méthodes heuristiques pour la construction de solutions réalisables. Nous disons que chercher une solution réalisable pour le problème du MMKP est lui-même un problème NP-difficile.

2.4.2 Une méthode de séparation et évaluation pour le MMKP

Dans cette section, nous présentons une méthode de résolution exacte pour le MMKP : méthode de séparation et évaluation. Celle-ci produit une solution optimale pour des instances pouvant contenir jusqu'à 300 éléments répartis sur 10 classes (voir Khan *et al.* [27]).

Pour chacun des états de la solution du MMKP, une classe est dite *libre* si aucun élément n'a été fixé dans cette classe, ou qu'elle est dite une classe *fixée* si au contraire, un élément a été fixé dans cette classe. L'état *fixée/libre* des classes est désigné par un vecteur s. Si une classe i est libre, $s_i = 0$, sinon la classe est fixée et $s_i = 1$. Nous représentons, l'état de la solution d'un nœud par un vecteur $\{x_{ij}\}$, $(i = 1, \ldots, n)$, $(j = 1, \ldots, r_i)$. Pour une classe fixée i, si $x_{ij} = 1$, ceci signifie que l'élément j de la i-ème classe est choisi dans la solution, et $x_{ij} = 0$ sinon.

Une borne supérieure pour le MMKP

Supposons qu'au nœud t, on a n_f classes fixées et $n_\ell = n - n_f$ classes libres. La valeur de la fonction objectif calculée en ce nœud est obtenue depuis les classes fixées de sorte qu'on a :

$$V(t) = \sum_{i=1}^{n} \sum_{j=1}^{r_i} (s_i = 1) v_{ij} x_{ij}.$$

La capacité restante des classes libres après fixation des classes de branchement est obtenue comme suit :

$$C_\ell^k = C^k - \sum_{i=1}^{n} \sum_{j=1}^{r_i} (s_i = 1) w_{ij}^k, \ k = 1, \ldots, m.$$

Considérons à présent l'instance $I_2(t)$ du MMKP. Il s'agit de l'instance telle qu'il reste $n_\ell = n - n_f$ classes libres à fixer de l'instance de départ I et le nouveau vecteur capacité $C_\ell = (C_\ell^1, C_\ell^2 \ldots, C_\ell^m)$ correspondant aux capacités restantes après fixation.
On définit la borne supérieure UB du MMKP à un nœud t comme suit :

$$UB(t) = V(t) + V_{LP(I_2(t))},$$

où LP(I_2(t)) représente le problème relâché de l'instance $I_2(t)$ issue de l'instance initiale I et $V(.)$ étant l'évaluation de la fonction objectif $Z(x)$.
Puisque $V_{LP(I_2(t))} \geq V_{I_2(t)}$, $UB(t)$ est donc une borne supérieure de I atteinte depuis le nœud t. Et comme solution efficace pour le problème $LP(I)$, on utilise la méthode du simplexe par exemple.

Un algorithme optimal

Dans ce paragraphe, nous présentons les principales étapes de l'algorithme de séparation et évaluation tel qu'il a été proposé par Khan *et al.* [27]. Donnons une

représentation d'une solution de MMKP.

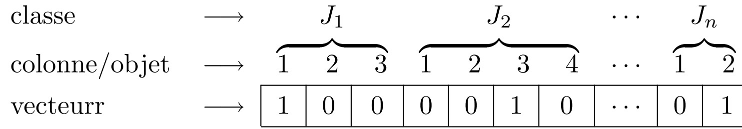

FIG. 2.4 – Représentation binaire d'une solution de MMKP.

Nous donnons d'abord quelques notations propres à cet algorithme.

Notations :

n : nombre de classes,
r_i : nombre d'éléments dans la classe i,
v : vecteur valeur solution,
W_{ij} : vecteur poids d'un élément j de la classe i,
C : vecteur capacité, x : vecteur solution,
s vecteur état des classes,
C_ℓ : vecteur capacité restante après fixation,
V : valeur de la fonction objective $Z(x)$,
n_f nombre de classes fixés et n_l nombre de classes libres,
b : classe du branchement,
UB : borne supérieure pour le MMKP.

L'algorithme que nous décrivons dans la suite fonctionne de la manière suivante :

1. On démarre par une solution dont l'état de toutes les classes est libre, ensuite on calcule une borne supérieure à l'aide de la méthode du simplexe calculée par le `Cplex` par exemple. On choisit la classe b de branchement. Il s'agit d'une classe dont l'état est libre. On initialise la recherche par ce nœud comme nœud actif de démarrage.

2. On cherche un nœud e appelé *e-nœud*, celui-ci est le nœud actif ayant la plus grande borne supérieure calculée UB.

3. Si le nœud e ne contient aucune classe libre (c'est-à-dire toutes les classes sont fixées : $(s_i = 1, \ \forall i = 1, \dots, n)$), alors le nœud e représente la solution optimale et l'algorithme s'arrête.

4. Si le nœud e possède au moins une classe libre i ($s_i = 0$), alors ce nœud est développé en fixant la classe de branchement b. Sélectionner la classe b revient à réaliser les étapes suivantes pour chaque élément de la classe b :
   - construire un nœud t où tous les éléments fixés sont les éléments fixés du nœud e et l'élément j de la classe b,
   - calculer la borne supérieure UB en ce nœud,
   - choisir la classe de branchement s'il existe des classes encore libres en ce nœud,
   - si le nœud t est réalisable, alors insérer ce nœud comme nœud actif dans l'arbre de recherche.
5. Aller à l'étape 2.

Ci-après les principales étapes de l'algorithme décrit dans la figure 2.5 :

```
Entrée : Une instance de MMKP ;
Sortie : Solution optimale de MMKP ;
Initialisation :
1.   V = 0, n_ℓ = n, C_ℓ^k = C^k ∀k = 1,...,m, s_i = 0, ∀i = 1,...,n ; /* Initialiser avec toutes les classes libres */
2.   UB ⟵ Simplexe() ; /* Trouver une borne supérieure pour MMKP*/
3.   x_{bj'} ⟵ max{x_ij : s_i = 0, i = 1,...,n, j = 1,...,r_i} ; /*x_ij ∈ [0,1], solution réelle trouvée par Simplexe()*/
4.   Insernœud(x, s, V, n_ℓ, C_ℓ, b, UB) ; /* Insérer le 1er nœud dans l'arbre */
Etape générale :
Tant que(1)
1.   t ⟵ ExtrairenœudMax() ; /* Extraire le nœud actif ayant la plus grande valeur UB */
2.   x = t → x, s = t → s, b = t → b /* Développement de la branche b */
3.   Si(t → n_2 ≤ 0) alors
       retourner t → x ; /* Retour s'il n'y a plus de variables libres*/
4.   s_b = 1, x_b = 0, n_ℓ = (t → n_ℓ) − 1 ; /* Fixation de la classe de branchement b */
5.   Pour j = 1,...,r_b faire
       y = x, C_ℓ= t → C_ℓ ;
       y_bj = 1 ; /* Fixation de l'élément j de la classe b */
       C_ℓ^k=C_ℓ^k − W_bj^k, ∀k = 1,...,m, V = (t → V) + v_bj ; /* Mise à jour de C_ℓ et de V */
       UB ⟵Simplexe() ; /* Trouver une borne supérieure UB */
       Si (UB > 0) alors
         x_{b'j'} ⟵max{y_ij :s_i = 0, i = 1,...,n, j = 1,...,r_i} ; /* Trouver la classe du branchement suivant */
         Insernœud(x, s, V, n_ℓ, C_ℓ, b', UB) ; /* Insérer le nœud actif suivant */
       Fin_Si
     Fin_Pour
Fin_Tant Que
```

FIG. 2.5 – Algorithme de résolution par séparation et évaluation pour le MMKP.

2.4.3 Méthodes de résolution approchée pour le MMKP

Il existe dans la littérature peu de méthodes heuristiques développées pour le MMKP. Moser *et al.* [40] ont proposé une heuristique de résolution basée sur une méthode de relaxation lagrangienne ou plus précisément sur une dégradation avantageuse des multiplicateurs de Lagrange. Une autre heuristique a été proposée par Khan *et al.* [28], cette dernière s'inspire du principe d'agrégation des ressources consommées et utilisé par Toyoda [56] pour résoudre le MDKP.

Une première heuristique pour le MMKP

A présent, nous présentons une première heuristique de résolution pour le MMKP. Celle-ci a été développée par Moser *et al.* [40].

L'heuristique, Heu1, proposée par Moser *et al.* [40] est antérieure à l'heuristique développée par Magazine et Oguz [33]. Cependant, elle est développée de telle sorte qu'elle prend en compte les contraintes du choix.
Cette heuristique, démarre en choisissant pour chaque classe J_i pour $i = 1, \ldots, n$, l'élément j du plus grand profit v_{ij} et en initialisant les différents multiplicateurs μ_k pour $k = 1, \ldots, m$ à zéro. En général, les contraintes de capacité seront violées.
Le choix initial des éléments sélectionnés est adapté pour respecter les contraintes de capacité. Ce procédé d'amélioration est répété pour la contrainte k' la plus violée parmi toutes les contraintes $k = 1, \ldots, m$ comme suit :

1. Pour chaque élément j de la classe J_i, on calcule l'accroissement δ_{ij} du multiplicateur $\mu_{k'}$. Cet accroissement résulte de l'interchange de l'élément sélectionné avec un autre élément j de la même classe.
2. Éventuellement, l'élément j' correspondant au plus petit accroissement pour le multiplicateur $\mu_{k'}$ est sélectionné pour l'interchange. Ce choix minimise la différence entre la solution optimale et la solution obtenue par l'heuristique Heu1. Ce procédé est réitéré jusqu'à ce que pour chaque classe, un élément ait été choisi sachant que les contraintes capacités sont vérifiées ou qu'il n'y ait plus possibilité d'interchange entre les éléments. Dans le premier cas, la solution obtenue est réalisable, dans le dernier cas, le problème n'admet pas de solution.
3. après que les étapes de Heu1, citées ci-dessus, aient été exécutées avec succès, il reste des capacités résiduelles du sac. Cet espace résiduel, doit être utilisé pour

améliorer la solution en remplaçant des éléments sélectionnés par des éléments de plus grand profit.
Chaque élément j de la classe J_i est comparé à l'élément sélectionné dans la solution en cours. On teste quel élément de plus grand profit peut remplacer l'élément fixé dans la solution et sans violer les contraintes capacités. Ce processus est réitéré jusqu'à ce qu'il n'y ait plus d'interchanges possibles. La solution obtenue par les éléments sélectionnés est réalisable.

Dans ce qui suit, nous présentons les principales étapes de l'algorithme Heu1 :

Entrée : V_{ij} : profit de l'objet j de la classe J_i pour $i = 1, \ldots, n$.
W_{ij}^k : k-ième poids de l'objet j de la classe J_i pour $i = 1, \ldots, n$.
J_i : classes d'objets j pour $i = 1, \ldots, n$.
C^k : capacité du sac pour $k = 1, \ldots, m$;
Sortie : Eléments sélectionnés de poids W_{ij}^k, pour $k = 1, \ldots, m$ $i = 1, \ldots, n$, $j = 1, \ldots, r_i$;

Initialisation et normalisation :
1. Pour $(k = 1, \ldots, m)$, faire $\mu_k = 0$; /* Démarrer avec des multiplicateurs nuls */
2. $\forall i = 1, \ldots, n$, Trouver l'indice j_i de l'élément de plus grand profit V_{ij} dans la classe J_i ;
Sélectionner cet élément ;
/* Sélectionner l'élément de plus grand profit dans chaque classe */
3. Pour chaque contrainte $k = 1, \ldots, m$, poser $\frac{W_{ij}^k}{C^k} = W_{ij}^k$, pour chaque $i = 1, \ldots, n$ $j = 1, \ldots, r_i$;
/* Normaliser les poids de chaque objet et par rapport à chaque contrainte*/,
4. Calculer $\Psi_k = \sum_{i=1}^{n} \sum_{j=1}^{r_i} W_{ij}^k$, pour $k = 1, \ldots, m$; /* Calculer la violation des contraintes */

Etape générale :
Tant que($\Psi_k \geq 1$, $k = 1, \ldots, m$ ou aucun inter change n'est possible) faire
1. Trouver la plus forte violation k' telle que $\Psi_{k'} > 1$;
2. $\delta_{ij} = (V_{ij_i} - V_{ij} - \sum_{k=1}^{m} \mu_{k'}(W_{ij_i}^k - W_{ij}^k))/(W_{ij_i}^{k'} - W_{ij}^{k'})$ pour $i = 1, \ldots, n$, $j = 1, \ldots, r_i$;
Trouver la classe i' et l'élément $j' \in J_{i'}$ correspondant au plus petit δ_{ij} ;
3. $\mu_{k'} = \mu_{k'} + \delta_{ij_i'}$ et $\Psi_k = \Psi_k - W_{ij_{i'}}^k + W_{ij_{i'}'}^k$, pour $k = 1, \ldots, m$;
4. Remplacer l'élément d'indice $j_{i'}$ de la classe $J_{i'}$ par celui d'indice $j_{i'}'$;
5. Calculer δ_{ij} pour tous les éléments non sélectionnés par rapport
à l'élément sélectionné dans chaque classe :
$\delta_{ij} = V_{ij_i} - V_{ij}$, si $V_{ij_i} - V_{ij} > 0$ et $\Psi_k = \Psi_k - W_{ij_{i'}}^k + W_{ij_{i'}'}^k \leq 1$, pour $k = 1, \ldots, m$
et $\delta_{ij} = 0$ sinon ;
6. Trouver la classe d'indice i' et l'élément d'indice $j_{i'}'$ de plus fort δ_{ij} ;
7. Faire l'inter change avec le nouvel élément ;
8. Calculer $\Psi_k = \sum_{i=1}^{n} \sum_{j=1, j \neq j_i'}^{r_i} W_{ij}^k + W_{ij_i'}^k$, pour $k = 1, \ldots, m$;
9. Si $\Psi_k \leq 1$, pour $k = 1, \ldots, m$ alors le problème admet une solution ;
Sinon le problème n'admet pas de solution ;
Fin_Tant Que

FIG. 2.6 – Procédure Heu1 : une première heuristique pour le MMKP.

L'heuristique Heu1 décrite dans la figure 2.6 a une complexité de l'ordre de $O(m(nr)^2 + mr)$ où $r = max\{r_1, \ldots, r_n\}$ (voir Moser *et al.* [40]).

Une deuxième heuristique pour le MMKP

Nous présentons ci-après la méthode de résolution approchée développée par Khan *et al.* [28]. Celle-ci est résumée dans les étapes suivantes :

1. L'algorithme démarre par une solution de sorte que dans chaque classe J_i, on fixe l'élément j de plus faible profit v_{ij}. Itérativement, on tente d'améliorer graduellement la solution obtenue, en remplaçant les éléments de plus faible profit par ceux de profit plus grand tout en garantissant la réalisabilité.
2. On utilise le concept d'agrégation des ressources de Toyoda [56] en transformant le vecteur poids d'un élément j de J_i, c'est-à-dire, $W_{ij} = (w_{ij}^1, \ldots, w_{ij}^m)$, comme suit :

$$a_{ij} = \frac{\langle W_{ij}, C \rangle}{|C|}, \quad \text{pour } j = 1, \ldots, |J_i|,$$

   où $\langle ., . \rangle$ est le produit scalaire dans $I\!R^m$ et $|.|$ est la norme euclidienne dans $I\!R^m$.

   L'idée principale est de pénaliser la consommation des ressources 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.
3. Par la suite, un processus de fixation d'un élément est appliqué. Celà 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.

Cette procédure, que nous notons par AppAlg, suppose que pour chaque classe J_i, les éléments j, pour $j = 1, \ldots, r_i$ sont rangés dans l'ordre croissant des valeurs profits, c'est-à-dire :

$$\text{si } j^{''} < j^{'} \text{ alors } \; v_{ij''} \leq v_{ij'}, \;\; \text{pour une classe } \; J_i.$$

La stratégie qui consiste à remplacer un élément de plus faible profit par un autre de profit plus important et tous deux appartenant à la même classe, s'appelle une

revalorisation puisqu'on tente d'augmenter la valeur de la solution. Une revalorisation est dite réalisable si la solution obtenue est réalisable, sinon la revalorisation est dite non réalisable.

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

Notations :

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

C_1 : vecteur des ressources consommées.

Δa : ressource agrégée épargnée.

Δv : gain total par unité de ressources agrégées.

Entrée : Une instance de MMKP ;
Sortie : Une solution approchée pour MMKP ;

Initialisation :

1. **Pour** $(i = 1, \ldots, n)$, **faire** $p_i = 1$; /* Démarrer avec des éléments de plus petite valeur profit */
2. $\forall k = 1, \ldots, m,\ C_1^k = \sum_{i=1}^{n} \sum_{j}^{r_i} w_{ip_i}^k$; /* Calculer les consommations des ressources*/

Étape générale :

Tant que(1) /* Amélioration itérative*/
1. **Pour** $(i = 1, \ldots, n)$, **faire** $\Delta a_{imax} = 0,\ \Delta v_{imax} = 0$;
2. **Pour** $(i = 1, \ldots, n;\ \ j = p_i + 1, \ldots, r_i)$ **faire**
   Si$(\exists k : k = 1, \ldots, m,\ C_1^k - w_{ip_i}^k + w_{ij}^k > C^k$, **continue** ;
   $\Delta a_i = \frac{\langle (W_{ip_i} - W_{ij}), C \rangle}{|C|}$; /* Calculer l'épargne des ressources agrégées */
   Si $(\Delta a_i > \Delta a_{imax})$ **alors**
      $\Delta a_{imax} = \Delta a_i,\ imax = i,\ j_{imax} = j$;
      /* Revaloriser avec le maximum de l'épargne des ressources */
   Si $(\Delta a_i < 0)$ **alors**
      $\Delta v_i = \frac{v_{ip_i} - v_{ij}}{\Delta a_i}$;
      Si $(\Delta v_i > \Delta v_{imax})$ **alors**
         $\Delta v_{imax} = \Delta v_i,\ imax = i,\ j_{imax} = j$;
      /* Revaloriser avec le maximum de valeur/agrégation */
   Fin_Si
  Fin_Pour
3. **Si** $\forall i = 1, \ldots, n$ $(\Delta a_{imax} < 0$ et$\Delta v_{imax} < 0)$;
   retourner $p_i,\ \forall i = 1, \ldots, n$;
4. $C_1^k = C_1^k - w_{imax p_{imax}}^k + w_{imax j_{imax}}^k$; /* Mise à jour des ressources consommées */
5. $p_{imax} = j_{imax}$;
  Fin_Pour
Fin_Tant Que

FIG. 2.7 – Procédure AlgApp : une deuxième heuristique pour le MMKP.

La boucle de "`Étape générale`" de AppAlg 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 :

1. trouver l'agrégation des ressources Δa de toutes les classes.
2. 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 ;
3. 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 $r = max\{r_1, \ldots, r_n\}$. En effet, AppAlg démarre en fixant les objets, dans chaque classe de plus faible profits v_{ij}, et le processus est réitéré jusqu'à obtention d'une première solution réalisable. Pour la classe i, AppAlg applique au plus $(r-1)$ revalorisations. Alors, le nombre maximum de revalorisations possibles est de $\sum_{i=1}^{n}(r_i - 1) \leq n(r-1)$. Ce qui signifie que la boucle générale de la revalorisation nécessite au maximum $n(r-1)$ opérations.
Maintenant, chaque itération de cette même boucle, utilise deux boucles supérieures avec deux étapes les plus coûteuses avec une complexité au pire chacune de $O(m)$. Donc la complexité des ces deux boucles est de $O(mn(r-1))$. En combinant les deux résultats, on conclut que AppAlg a une complexité au pire de $O(mn^2(r-1)^2)$ (voir [28]).

Chapitre 3

Un algorithme approché pour le problème de la distribution équitable

Nous traitons dans ce chapitre le problème de la distribution équitable ("Knapsack Sharing"). Nous proposons un algorithme de résolution approchée exploitant certaines caractéristiques de la recherche tabou. Dans un premier temps, nous présentons la version simple de cet algorithme dans laquelle nous utilisons (i) un paramètre profondeur limitant la recherche des solutions voisines autour des éléments dits "critiques" et (ii) une mémoire (liste tabou) permettant la non stagnation de certaines solutions. Dans un deuxième temps, nous proposons une version modifiée de l'algorithme en introduisant principalement une nouvelle profondeur. ([1])

3.1 Introduction

Nous rappelons que le problème de la distribution équitable (KSP) est défini par un ensemble de n objets et un ensemble de m classes, où à chaque objet j sont associés un

[1]Cette partie traite d'un article que nous avons publié : M. Hifi, S. Sadfi and A. Sbihi sous le titre *"An Efficient Algorithm for the Knapsack Sharing Problem"*, dans Computational Optimization and Applications, 2002 ; 23 :27-45, et d'une conférence internationale ; Conférence Internationale en Recherche Opérationnelle CIRO'02, UCAM, Marrakech, Maroc, 2002

profit v_j et un poids w_j. Si J_i désigne la i-ème classe d'objets, pour $i = 1, \ldots, m$, alors $\forall p = 1, \ldots, m,\ \forall q = 1, \ldots, m,\ p \neq q,\ J_p \cap J_q$ et $\cup_{i=1}^{m} J_i = \mathcal{N}$. L'objectif de ce problème est de déterminer le sous-ensemble d'objets à retenir dans le sac ("knapsack") dont la capacité a pour valeur c. De plus, ce sous-ensemble est choisi de manière à maximiser la valeur de la fonction objectif qui réalise le minimum par rapport à l'ensemble des classes de $\mathcal{N}$ sans violation de la capacité.

Formellement, le KSP peut être écrit sous la forme suivante :

$$(KSP)\begin{cases} \text{Maximiser} & \min\limits_{1\leq i\leq m} \left\{ \sum\limits_{j\in J_i} v_j x_j \right\} \\ s.c. & \sum\limits_{j\in\mathcal{N}} w_j x_j \leq c \\ & x_j \in \{0,1\},\ \text{pour } j = 1, \ldots, n. \end{cases}$$

où x_j dénote la variable de décision et qui vaut 1 si l'objet j est mis dans le sac, 0 sinon. Pour ce problème, nous rappelons aussi les hypothèses suivantes (énoncées dans le chapitre 2) :

- Les valeurs de $v_j,\ j \in \mathcal{N}$, sont des entiers strictements positifs.
- $\sum\limits_{j\in\mathcal{N}} w_j > c$, ce qui permet d'éliminer des cas triviaux.

Le KSP à variables binaires a été étudié par Yamada *et al.* [62, 63] et Hifi et Sadfi [22]. Ces auteurs se sont principalement intéressés à la résolution exacte du problème. Les méthodes exactes proposées s'appuient principalement sur (i) des méthodes de séparation et évaluation et (ii) des techniques de la programmation dynamique. Les auteurs ont montré que ces méthodes arrivaient à produire des solutions optimales pour le problème étudié, il n'en reste pas moins que pour traiter des instances plus importants du problème, il est nécessaire d'utiliser des méthodes approchées.

Dans ce chapitre, nous nous intéressons à la résolution approchée du problème de la distribution équitable. Dans un premier temps, nous présentons une première approche dans laquelle nous utilisons (i) un paramètre profondeur limitant la recherche des solutions voisines autour des éléments dits "critiques" et (ii) une mémoire (liste tabou) permettant la non stagnation de certaines solutions. Notons que cette approche s'inspire de la recherche tabou [14, 15, 16]. Par la suite, nous présentons une amélioration de l'algorithme de la première partie en introduisant une stratégie combinant l'intensification de la recherche et la diversification de la solution. Il s'agit principalement

d'introduire une modification sur la façon d'exploiter le paramètre profondeur. Finalement, nous présentons une partie expérimentale pour évaluer la performance des deux versions de l'algorithme.

3.2 Une première méthode approchée

Dans cette section, nous commençons (section 3.2.1) par décrire les éléments critiques associés à une solution du KSP. Par la suite (section 3.2.2), nous donnons une représentation d'une solution réalisable qui facilitera, lors de l'utilisation d'une recherche locale, l'utilisation de la notion d'une solution voisine. Dans la section 3.2.3 nous présentons une procédure constructive qui permet de générer une première solution réalisable pour le KSP. Finalement, dans la section 3.2.4, nous décrivons les étapes principales de la première version de l'algorithme.

3.2.1 Les éléments critiques et le problème de la distribution équitable

Nous utilisons une décomposition du problème KSP en une série de problèmes de sac-à-dos utilisée par Hifi et Sadfi [22]. Les auteurs ont montré que chacun des sac-à-dos, noté $KP_{J_i}^{\bar{c}_{J_i}}$, $i = 1, \ldots, m$, pouvait représenter un problème auxiliaire au problème du KSP. En effet, cette série de problèmes de sac-à-dos peut être décrite comme suit :

$$(KP_{J_i}^{\bar{c}_{J_i}}) \begin{cases} \text{Max} & \sum_{j \in J_i} v_j x_j \\ \text{s.c} & \sum_{j \in J_i} w_j x_j \leq \bar{c}_{J_i} \\ & x_j \in \{0,1\}, \text{ pour } j \in J_i \end{cases}$$

où $\bar{c}_{J_i}$ est un entier positif vérifiant $\bar{c}_{J_i} \leq c$, pour $i = 1, \ldots, m$, et $KP_{J_i}^{\bar{c}_{J_i}}$ représente le problème de sac-à-dos, de capacité $\bar{c}_i$, associé à la classe J_i.

Nous rappelons qu'un problème de sac-à-dos peut disposer d'un élément critique (voir le chapitre 2). Dans notre étude, nous considérons que chaque problème $KP_{J_i}^{\bar{c}_{J_i}}$, $i = 1, \ldots, m$, dispose d'un élément critique, noté r_{J_i} (lorsque les contraintes sur les

x_j sont relachées, i.e., $0 \leq x_j \leq 1$). Une adaptation des notations de la section ?? du chapitre 2 sur l'élément critique d'une solution du problème KP, permet de la réécrire de la manière suivante :

$$\sum_{k_{J_i}=1}^{r_{J_i}-1} w_{k_{J_i}} \leq \bar{c}_{J_i} \tag{3.1}$$

$$\sum_{k_{J_i}=1}^{r_{J_i}} w_{k_{J_i}} > \bar{c}_{J_i} \tag{3.2}$$

$$\sum_{i=1}^{m} \bar{c}_{J_i} \leq c \tag{3.3}$$

L'équation (3.1) signifie que tous les éléments, dont l'indice est inférieur à celui de l'élément critique r_{J_i}, sont fixés à "un" et par conséquent, la solution ainsi obtenue reste réalisable pour cette classe. L'équation (3.2) signifie que le rang de l'élément critique pour lequel la solution en cours devient non réalisable est l'élément lui correspondant fixé à "un". Finalement, l'équation (3.3) entraine que la solution reste réalisable pour chaque sac-à-dos et donc réalisable pour le KSP.

3.2.2 Représentation de la solution

Nous avons choisi une représentation standard pour le problème KSP qui consiste à considérer la configuration sous forme d'un vecteur S (voir la figure 3.1) de dimension n. Le contenu de chaque case $S(j) = x_j$ du vecteur désigne si l'objet j est fixé à 0 ou à 1 dans la solution.

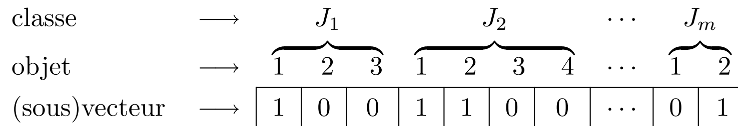

FIG. 3.1 – Représentation d'une solution du KSP.

Cette représentation permet aussi de faciliter le calcul de l'évaluation de chaque

solution, donnée par :

$$f(S) = \min \Big\{ \sum_{j \in J_1} v_j S(j), \ldots, \sum_{j \in J_m} v_j S(j) \Big\}, \quad \text{où} \quad S = (S_1, \ldots, S_m).$$

Notons que, dans notre cas, des précautions peuvent être prises afin d'éviter les solutions non réalisables (comme nous le décrirons par la suite). Différents auteurs (voir par exemple Chu et Beasley [6]) ont aussi appliqué d'autres représentations (ou équivalentes) qui semblaient pertinentes pour certains problèmes de l'optimisation combinatoire. Parmi ces représentations, nous pouvons adapter un des cas de figure suivants :

1. l'utilisation d'une représentation permettant de passer d'une solution réalisable à une autre (elle assure automatiquement la réalisabilité de chaque solution) ;
2. la séparation de la fonction d'évaluation en deux termes : un premier terme contenant un sous-ensemble représentant une solution réalisable pour le problème et un deuxième terme représentant la non réalisabilité de la solution ;
3. concevoir une procédure permettant de transformer toute solution obtenue (pas nécessairement réalisable) en une solution réalisable.

Dans notre étude, nous avons appliqué une autre approche qui s'appuie sur la notion des *éléments critiques*. Nous considérons qu'un élément critique d'une classe donnée est l'élément qui sépare la classe en deux régions : une première région dite *région de gauche* et une deuxième appelée *région de droite*. Ce procédé permet d'assurer la réalisabilité de la (nouvelle) solution obtenue. En effet, l'application des étapes suivantes permet à tout moment de générer une solution réalisable pour le KSP :

1. considérer une partie de chaque (sous)vecteur S_i, $i = 1, \ldots, m$, et soit $S_i(k)$, $k \leq |J_i|$, l'*élément critique* de la i-ème classe ;
2. fixer tous les éléments de la région critique de gauche de chaque classe à "`un`".
3. fixer les éléments de la région critique de droite de chaque classe à "`zéro`".

Les trois étapes (1)-(3) permettent de construire une solution réalisable. Par ailleurs, on peut remarquer qu'une meilleure solution (de meilleure qualité) peut être obtenue en fixant à "`un`" certains éléments de la région critique de droite.

Dans notre étude, nous considérons que les éléments de la région de droite sont supposés "libres", i.e. qu'à tout moment on peut appliquer une procédure sur ces éléments afin de tenter d'améliorer la qualité de la solution en cours.

3.2.3 Construction d'une solution initiale

Dans cette section, nous proposons une procédure gloutonne permettant de générer une première solution réalisable pour le KSP. D'une manière générale, cette procédure (notée GH) tente de déterminer un ensemble d'éléments critiques (comme nous l'avons décrit auparavant dans la section 3.2.1) en prenant en compte des critères de préférences sur certaines classes. Cette façon de construire la solution permet de favoriser les éléments réalisant le plus fort rapport *profit* par *poids*, i.e. $v_j/w_j,\ j \in \mathcal{N}$, en altérnant entre les différentes classes.

```
Entrée : Une instance du KSP;
Sortie : Une solution réalisable S et son évaluation O(S);
-----------------------------------------------------------
1.  Poser SumCap ⟵ 0;
    /* initialiser la capacité totale cumulée */
2.  Poser min ⟵ 1,;
    /* choix de la classe j_min d'évaluation minimum */
3.  Pour i ∈ {1, ..., m}, j_i ⟵ 1, V_i ⟵ 0 et W_i ⟵ 0;
    /* V_i : (resp. W_i : ) valeur cumulée (resp. le poids ) des j choisis dans J_i */
Répéter
  Si SumCap + w_{j_min} ≤ c alors
    Poser :
     - SumCap ⟵ SumCap + w_{j_min};
     - V_min ⟵ V_min + v_{j_min};
     - W_min ⟵ W_min + w_{j_min};
     - j_min ⟵ j_min + 1 et min ⟵ argmin_{1≤i≤m} {V_i};
Jusqu'à j_min > |J_min|.
```

FIG. 3.2 – La procédure GH : construction d'une (première) solution réalisable.

La procédure GH (voir Figure 3.2) permet, d'une façon gloutonne, de générer une première solution réalisable. L'idée consiste (i) à sélectionner une classe et (ii) à fixer une composante de la classe choisie à `zéro` ou à `un` en suivant certains critères. Ce procédé est réitéré jusqu'à fixation de tous les éléments.

Nous rappelons qu'initialement les éléments de chaque classe sont ordonnés par

ordre décroissant du rapport *profit* par *poids*. A chaque étape de la boucle `répéter` (voir Figure 3.2), une classe d'indice min est sélectionnée. Le choix imposé sur les classes permet de fixer définitivement, par la suite, des éléments à `un` tout en essayant d'effectuer un équilibre entre les évaluations des classes (puisqu'il s'agit d'un problème en $max - min$). En effet, un indice (noté j_{min}) de la classe choisie est fixé à "`un`" (ajouté au sac) si (i) l'élément est libre et (ii) son poids ne dépasse pas la capacité résiduelle restante. Dans le cas contraire, l'élément est fixé à "`zéro`". Ce processus est répété jusqu'à saturation du sac ou bien fixation de tous les éléments.

3.2.4 L'algorithme

Nous décrivons, dans cette partie, les étapes principales de l'algorithme. L'idée de cette approche consiste à partir d'une solution réalisable du KSP et de tenter de l'améliorer en appliquant une *recherche complémentaire*. Il s'agit (i) de construire une solution partielle réalisable et (ii) de la compléter par la suite en appliquant la procédure GH décrite dans la section 3.2.3. Nous allons donc définir le *voisinage* d'une solution puis décrire le *processus de recherche* permettant de générer au fûr et à mesure une ou des solutions de meilleure qualité.

A) Description des phases principales de l'algorithme

Dans la section 3.2.1, nous avons représenté les éléments critiques d'une solution du KSP sous forme d'un ensemble fini de m éléments. Ces différents éléments correspondent aux différentes classes du problème. Par ailleurs, nous pouvons remarquer qu'à chaque ensemble d'éléments critiques correspond un ensemble de solutions réalisables. Dans notre étude, au lieu de travailler directement sur une solution réalisable, nous avons préféré travailler sur l'ensemble des éléments critiques représentant cette solution.

En effet, l'idée de l'approche consiste, dans un premier temps, à générer un certain nombre de solutions (voisines) qui correspondent à l'ensemble des éléments critiques en cours, puis de considérer un voisinage de l'ensemble des éléments critiques, dans un deuxième temps. Ce dernier ensemble correspondra à la meilleure évaluation du voisinage.

En partant d'une solution réalisable (représentant un ensemble d'éléments critiques

donné par $(r_{J_1}, \ldots, r_{J_m})$), l'approche procède en trois phases :

Phase 1.

Pour chaque classe J_i, $i = 1, \ldots, m$, déplacer l'élément critique r_{J_i} d'une unité vers la gauche. Dans ce cas, il s'agit de forcer l'élément $r_{J_i} - 1$ à "`zéro`" permettant à cet élément de devenir l'élément critique (cf. Figure 3.3) de la classe J_i. Par la suite, la solution est complétée (en reconsidérant tous ou certains éléments des différentes classes) par l'application de la procédure GH (cf. Figure 3.3 : pour la i-ème classe, les éléments libres, représentés par le symbole $*$, sont cette fois-ci fixées à "`zéro`" ou à "`un`"). Nous pouvons voir que l'application de la procédure GH permet aussi de changer la position des autres éléments critiques.

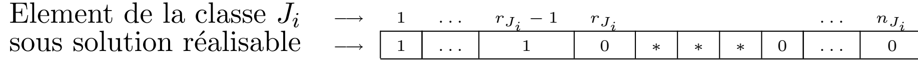

Fig. 3.3 – Représentation d'une solution partielle réalisable du KSP pour la classe J_i. Le symbole $*$ représente les éléments libres non encore fixés pour une classe donnée.

Phase 2.

La première phase est réitérée sur une profondeur donnée Δ pour chaque classe J_i (cf. Figure 3.3 : il suffit de remplacer la position de l'élément critique $r_{J_i} - 1$ par $r_{J_i} - \Delta$). Il s'agit d'une généralisation de la notion de déplacement d'un élément critique permettant de générer un voisinage plus important pour la solution de départ (ou pour l'ensemble des éléments critiques en cours).

Phase 3.

Soit *Sol* la structure de la meilleure solution générée par l'application des deux phases précédentes. Répéter, un certain nombre de fois, ce processus de recherche sur la meilleure solution *Sol* du voisinage. Nous rappelons que la meilleure solution retenue pour le KSP (ou pour répéter la recherche) est la solution réalisant l'évaluation minimum sur l'ensemble des classes.

La première version de l'algorithme s'appuie principalement sur le processus décrit par les trois phases précédentes. On peut remarquer que la phase 1 peut être considérée comme l'étape principale de l'algorithme, puisque les deux autres phases permettent juste d'effectuer un nouveau déplacement (changement de profondeur sur l'une des classes) puis relancer le processus de construction. Dans ce qui suit, nous allons donc expliquer les différentes étapes utilisées par la phase 1.

Dans un premier temps, nous pouvons considérer que le choix du déplacement n'est pas aléatoire, puisque souvent les solutions optimales du problème KP sont localisées aux frontières de l'élément critique (le core du sac-à-dos) (voir Horowitz et Sahni [26] et Pisinger [45]). Nous avons donc adapté ce principe au KSP tout en considérant, cette fois-ci, la généralisation sur un ensemble d'éléments critiques.
Dans un deuxième temps, nous pouvons remarquer que le déplacement d'un élément critique et l'application de la procédure GH permettent de construire une nouvelle solution réalisable. Cette dernière sera considérée comme une solution du voisinage de l'ensemble des éléments critiques en cours. En effet, la procédure GH agit de la manière suivante :

a) On suppose que les éléments (de toutes les classes) se positionnant sur la partie gauche de chaque élément critique (en prenant compte du nouvel élément critique $r_{J_i} - \Delta$) sont fixés à "`un`".

b) Appliquer la procédure GH sur le reste des éléments considérés comme "libres".

Les étapes (a) et (b) permettent de générer un nouvel ensemble d'éléments critiques ce qui permet aussi de donner une nouvelle solution réalisable pour le KSP. Notons qu'à chaque classe i est associé un paramètre profondeur Δ_i, pour $i = 1, \ldots, m$. Afin de traiter toutes les classes et pour ne pas avoir un débordement à gauche, on pose $\Delta_i = min\{\Delta, \Delta_i\}$.

```
Entrée  : Une instance du KSP et une profondeur Δ;
Sortie  : Une solution approchée de valeur MeilSol;

Poser Sol ⟵ GH(), MeilSol ⟵ Sol et Iter ⟵ 0;
Initialiser la liste tabou notée Tabou_Liste;
Répéter
    Poser Sol ⟵ Trouve_Meil_Sol_Voisinage(Sol, Δ);
    Mettre_à_Jour_Tabou(Sol, Tabou_Liste);
    Poser MeilSol ⟵ max{MeilSol, Sol};
    Incrémenter(Iter);
Jusqu'à Iter > MaxIter;
Sortir avec MeilSol;
```

FIG. 3.4 – La première version de l'algorithme : utilisation d'une profondeur.

B) Introduction d'une mémoire

Nous venons de voir que l'application d'un mouvement sur un élément critique permet de générer une nouvelle solution pour le KSP, représentée par un ensemble d'éléments critiques. L'application de ce mouvement est concrétisée par un déplacement de l'élément critique d'une classe d'une position (à une itération) vers la gauche. Par ailleurs, l'utilisation de la procédure GH peut aussi induire des mouvements des autres éléments critiques vers les régions critiques de droites.
La répétition de ce processus de construction peut, dans certains cas, générer la même configuration à des itérations différentes. Dans ce cas, on peut remarquer que l'interdiction de certains mouvements sur les éléments critiques peut espérer la génération d'un certain nombre de configurations différentes. Dans notre étude, cette interdiction est illustrée par l'introduction d'une mémoire, dite *liste tabou*, interdisant le retour en arrière d'un élément critique fixé.

Plusieurs représentations de la liste tabou peuvent être considérées. Dans notre étude nous avons considéré que la mémoire peut être représentée par un tableau de listes chaînées. Chaque liste chaînée représente une file, où chaque élément de la file

est caractérisé par un couple représentant le mouvement (les positions de déplacement) interdit. Cette représentation permet de traiter séparément les différentes classes, ce qui permet aussi d'éviter le parcours de tous les éléments du tableau.

De la même façon, nous avons limité l'interdiction de chaque mouvement tabou permettant au processus de recherche l'exploration d'autres solutions voisines. Dans note étude, nous avons considéré que le nombre de mouvements tabou est limité à une certaine taille (définie expérimentalement).

Par ailleurs, nous avons introduit un *critère d'aspiration* qui consiste à accepter une solution voisine même si cette dernière est obtenue par l'application d'un mouvement tabou (interdit). En effet, nous avons appliqué ce principe lorsque la qualité de la solution générée est meilleure que la meilleure évaluation trouvé jusqu'à présent par le processus de recherche.

Un résumé de la première version de l'algorithme est donné par la figure 3.4. La première étape de l'algorithme consiste à construire la première solution réalisable Sol et d'initialiser la liste tabou à vide ainsi que la sauvegarde de la meilleure solution en cours, notée $MeilSol$. La deuxième étape de l'algorithme est représentée par la boucle `Répéter` qui est composée par : (i) lancer le processus de recherche à partir d'une solution Sol, (ii) effectuer une mise à jour de la liste tabou et (iii) garder la meilleure solution en cours. Ce procédé est réitéré un nombre maximum de fois (noté $MaxIter$). Notons que le critère d'aspiration est appliqué lors de l'appel de la procédure `Mettre_à_Jour_Tabou`$(Sol, Tabou_Liste)$.

Complexité de la méthode approchée

L'algorithme a une complexité en $O(mn)$. D'une part, à chaque étape de la boucle `Répéter`, la procédure `Trouve_Meil_Sol_Voisinage`(Sol, Δ) prend $\Delta \times O(mn)$ où Δ est une constante représentant le paramètre profondeur. Par conséquent, pour un nombre maximum d'itérations $MaxIter$, on a $MaxIter \times \Delta \times O(mn)$ opérations. D'autre part, la procédure `Trouve_Meil_Sol_Voisinage`(Sol, Δ) possède la même complexité. Donc l'algorithme est en $O(mn)$.

3.3 Une deuxième méthode approchée

Dans cette partie, nous proposons une deuxième version de l'algorithme. Nous avons vu dans la section 3.2.4 comment utiliser une profondeur simple en partant d'une position donnée vers la gauche tout en effectuant des déplacements répétés. Nous proposons cette fois-ci l'utilisation de deux profondeurs qui agissent de manière différente. En effet, cette partie portera sur la description de la nouvelle version de l'algorithme. Dans la section 3.3.1, nous décrivons les différentes profondeurs utilisées et nous présentons la notion du *voisinage*. Dans la section 3.3.2 nous décrivons le *processus de recherche* appliqué et les étapes principales de l'algorithme.

3.3.1 Introduction d'une deuxième profondeur

Dans la section 3.2.1, nous avons représenté les éléments critiques d'une solution du KSP sous forme d'un ensemble fini de m éléments. Ces différents éléments correspondent aux différentes classes du problème.

En effet, l'idée de l'approche consiste, dans un premier temps, à générer un certain nombre de solutions (voisines) qui correspondent à l'ensemble des éléments critiques en cours, puis de considérer un voisinage de l'ensemble des éléments critiques, dans un deuxième temps. Ce dernier ensemble correspondra à la meilleure évaluation du voisinage.

Cette version peut être considérée comme une généralisation de la première version. Elle consiste à mieux exploiter le voisinage de la solution et de voir comment définir une meilleure orientation de la recherche. Dans ce qui suit, nous introduisons une profondeur à deux sens :

1. un parcours en profondeur à gauche, noté Δ_g permettant de limiter le voisinage ;
2. un parcours en profondeur à droite, noté Δ_d permettant de simuler la diversification.

L'utilisation des deux profondeurs s'applique comme suit :

(i) nous réalisons d'abord le parcours à gauche pour optimiser la localisation et pour limiter la taille du “meilleur voisinage”,

(ii) nous introduisons un parcours à droite afin de diversifier la recherche.

Nous considérons donc deux régions critiques. Initialement, nous fixons la région critique de gauche pour chaque classe J_i (pour limiter la taille de l'exploration) et sur laquelle nous appliquons une recherche tabou. Pour chaque parcours considéré, nous imposons à cette classe un parcours dit à droite qui consiste à diversifier le voisinage en explorant la région critique de droite et tout en essayant de localiser des solutions non encore visitées.

En partant d'une solution réalisable (représentant un ensemble d'éléments critiques donné par $(r_{J_1}, \ldots, r_{J_m})$), l'approche procède en trois phases :

Phase 1.

Pour chaque classe J_i, $i = 1, \ldots, m$, déplacer l'élément critique r_{J_i} d'une unité vers la gauche. Dans ce cas, il s'agit de forcer l'élément $r_{J_i} - 1$ à "`zéro`" permettant à cet élément de devenir l'élément critique (cf. Figure 3.5) de la classe J_i. A partir de cet élément critique, nous considèrons des déplacements à droite de $r_{J_i} + 1$ jusqu'à $r_{J_i} + \Delta_d$. Par la suite, la solution est complétée (en reconsidérant certains éléments des différentes classes) par l'application de la procédure GH (cf. Figure 3.5 : pour la i-ème classe, les éléments libres, représentés par le symbole $*$, sont cette fois-ci fixés à "`zéro`" ou à "`un`"). Notons que l'application de la procédure GH permet aussi de changer la position des autres éléments critiques.

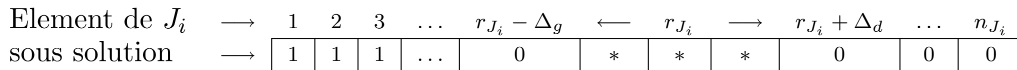

FIG. 3.5 – Représentation d'une solution partielle de KSP utilisant deux profondeurs pour une classe J_i.

Phase 2.

La première phase est réitérée sur un couple de profondeurs donné par (Δ_g, Δ_d) pour chaque classe J_i (cf. Figure 3.5 : il suffit de remplacer la position de l'élément critique $r_{J_i} - 1$ par $r_{J_i} - \Delta_g$). Pour chaque déplacement effectué à gauche de l'élément critique, nous explorons toute la région de droite en effectuant des déplacements successifs à droite de $r_{J_i} + 1$ jusqu'à $r_{J_i} + \Delta_d$. A chaque déplacement à droite, la solution

est complétée par l'application de la procédure GH. Il s'agit d'une généralisation de la notion de déplacement d'un élément critique permettant de générer un voisinage plus important pour la solution de départ (ou pour l'ensemble des éléments critiques en cours).

Phase 3.

Soit *Sol* la structure de la meilleure solution générée par l'application des deux phases précédentes. Répéter, un certain nombre de fois, ce processus de recherche sur la meilleure solution *Sol* du voisinage. Nous rappelons que la meilleure solution retenue pour le KSP (ou pour répéter la recherche) est la solution réalisant l'évaluation minimum sur l'ensemble des classes.

3.3.2 Le principe de l'algorithme

La deuxième version de l'algorithme s'appuie principalement sur le processus décrit par les trois phases précédentes. On peut remarquer que la phase 1 peut être considérée comme l'étape principale de l'algorithme, puisque les deux autres phases permettent juste d'effectuer un nouveau déplacement à gauche et des déplacements successifs à droite (changement de profondeurs sur l'une des classes) puis relancer le processus de construction. Dans ce qui suit, nous allons donc expliquer les différentes étapes utilisées par la phase 1.

Nous pouvons remarquer que pour chaque déplacement à gauche, les déplacements successifs à droite d'un élément critique et l'application de la procédure GH permettent de construire une nouvelle solution réalisable pour chacun de ces déplacements. Cette dernière sera considérée comme une solution du voisinage de l'ensemble des éléments critiques en cours. En effet, la procédure GH agit de la manière suivante :

a) On suppose que les éléments (de toutes les classes) se positionnant sur la partie gauche de chaque élément critique (en tenant compte du nouvel élément critique $r_{J_i} - \Delta_g$) sont fixés à "un" ainsi que les éléments se situant à gauche de l'élément critique après chaque déplacement à droite de $r_{J_i} + 1$ jusqu'à $r_{J_i} + \Delta_d$.

b) Appliquer la procédure GH sur le reste des éléments considérés comme "libres".

Les étapes (a) et (b) permettent de générer un nouvel ensemble d'éléments critiques, ce qui permet aussi de donner une nouvelle solution réalisable pour le *KSP*. Notons

qu'à chaque classe i est associé un couple de paramètres profondeurs $(\Delta_{g_i}, \Delta_{d_i})$ pour $i = 1, \ldots, m$. Afin de traiter toutes les classes et pour ne pas provoquer un débordement à gauche et/ou à droite, on pose $\Delta_{g_i} = min\{\Delta_g, \Delta_{g_i}\}$ et chaque valeur de Δ_{d_i} est au moins égale au double de Δ_{g_i}.
La version modifiée remplace la procédure `Trouve_Meil_Sol_Voisinage`(Sol, Δ) par la procédure utilisant deux profondeurs `Trouve_Meil_Sol_Voisinage`$(Sol, \Delta_g, \Delta_d)$.

Nous venons de voir que l'application d'un mouvement à deux profondeurs sur un élément critique permet de générer une nouvelle solution pour le KSP. Celle-ci est représentée par un ensemble d'éléments critiques. L'application de ce mouvement est concrétisée par un déplacement de l'élément critique d'une classe aussi bien dans la région critique de gauche que de droite (à une itération). L'utilisation de la procédure GH peut aussi induire des mouvements des éléments critiques des autres classes.
La réitération de ce processus de construction peut, comme dans la version à profondeur simple, générer la même configuration pour la solution à des itérations ultérieures. Dans ce cas, interdire certains mouvements sur les éléments critiques peut encourager la génération d'un certain nombre de configurations différentes.
D'une manière similaire à la première version, nous illustrons cette interdiction par l'introduction d'une mémoire sous forme d'une *liste tabou* interdisant le retour en arrière d'un élément critique fixé.

Complexité de la deuxième méthode approchée

L'algorithme a une complexité en $O(mn)$. Notons simplement qu'à chaque étape de la boucle `Répéter`, la procédure `Trouve_Meil_Sol_Voisinage`$(Sol, \Delta_d, \Delta_g)$ prend $\Delta_d \times \Delta_g \times O(mn)$ et pour un maximum nombre d'itérations elle est en $\Delta_d \times \Delta_g \times MaxIter \times O(mn)$, où $\Delta_d \times \Delta_g \times MaxIter$ est une constante. Par conséquent, la deuxième version de l'algorithme est aussi en $O(mn)$.

3.4 Partie expérimentale

Les différentes versions de l'algorithme ont été codées en langage `C` et exécutées sur une station SUN `UltraSparc10` (250 Mhz avec 128 Mo de RAM). Nous avons effectué notre étude expérimentale sur un ensemble composé de 240 instances de différentes

tailles et densités. Ces instances sont standards (détaillées dans le Tableau 3.1) et leurs solutions optimales sont connues et tirées de Hifi et Sadfi [22]). Ces instances sont réparties en deux catégories. Une première catégorie d'instances représentée par un groupe de 168 instances "non corrélées". Une deuxième catégorie regroupe 72 instances "fortement corrélées".

Inst.	n	m	Inst.	n	m	Inst.	n	m
A02.x	1000	2	B02.x	2500	2	C02.x	5000	2
D02.x	7500	2	E02.x	10000	2	F02.x	20000	2
A05.x	1000	5	B05.x	2500	2	C05.x	5000	2
D05.x	7500	5	E05.x	10000	2	F05.x	20000	2
A10.x	1000	10	B10.x	2500	10	C10.x	5000	10
D10.x	7500	10	E10.x	10000	2	F10.x	20000	10
A20.x	1000	20	B20.x	2500	20	C20.x	5000	20
D20.x	7500	20	E20.x	10000	20	F20.x	20000	20
A30.x	1000	30	B30.x	2500	30	C30.x	5000	30
D30.x	7500	30	E30.x	10000	30	F30.x	20000	30
A40.x	1000	40	B40.x	2500	40	C40.x	5000	40
D40.x	7500	40	E40.x	10000	40	F40.x	20000	40
A50.x	1000	50	B50.x	2500	50	C50.x	5000	50
D50.x	7500	50	E50.x	10000	50	F50.x	20000	50
A02C.x	1000	2	B02C.x	2500	2	C02C.x	5000	2
D02C.x	7500	2	E02C.x	10000	2	F02C.x	20000	2
A05C.x	1000	5	B05C.x	2500	5	C05C.x	5000	5
D05C.x	7500	5	E05C.x	10000	5	F05C.x	20000	5
A10C.x	1000	10	B10C.x	2500	10	C10C.x	5000	10
D10C.x	7500	10	E10C.x	10000	10	F10C.x	20000	10

TAB. 3.1 – Détail des instances de test : $1 \leq$x≤ 4.

Les instances "non corrélées" sont générées comme suit : pour chaque instance, le nombre de classes m est pris dans l'intervalle $[2, 50]$, le nombre de variables n est pris dans $[1000, 20000]$ et w_j et v_j sont mutuellement indépendants et uniformément générés dans $[1, 100]$ et $[1, 50]$, respectivement. La capacité c est fixée à $\lfloor (\sum_{j=1}^{n} w_j)/2 \rfloor$ et, la cardinalité de chaque classe $J_i,\ i = 1, \ldots, m$, est prise dans l'intervalle $[1, n - m + 1]$. Les autres 72 instances "corrélées" sont générées comme les "instances non corrélées", à la différence que les valeurs des profits v_j, associées aux éléments, sont égales à $w_j + 100$, pour $j = 1, \ldots, n$.

3.4.1 Sommaire des résultats

Dans nos expériences numériques, nous avons considéré les différentes versions de l'algorithme. Les résultats obtenus pour les deux groupes d'instances, sont reportés dans le Tableau 3.2. Nous avons reporté, pour chaque version de l'algorithme, le nombre d'instances résolues à l'optimum (noté *Opt*) et le pourcentage de déviation moyenne (*Av. P.D.*) donné par :

$$Av.\ P.D. = \sum_{i=1}^{\ell} \Big(\frac{Opt_i - A_i}{Opt_i} \times 100\% \Big) / \ell$$

où ℓ représente le nombre d'instances traitées, A_i représente l'évaluation de la solution obtenue en appliquant l'algorithme sur l'instance i et, Opt_i est l'évaluation de la solution optimale de cette instance. Nous avons également reporté le pourcentage de déviation au pire de chaque classe, noté $Worst$.

Initialement, nous avons considéré une version simple combinant la procédure GH et une recherche locale simple, appelée GH-LS. Dans ce cas, nous avons supprimé la liste mémoire. La première version de l'algorithme, appelée STS, représente l'algorithme GH-LS avec une liste mémoire. En dernier lieu, nous avons considéré la deuxième version de l'algorithme notée MTS.
L'implémentation des différentes versions de l'algorithme nécessite quelques réglages : la façon de fixer le nombre d'itérations, la meilleure valeur de la profondeur de gauche Δ_g (ou Δ), celle de la profondeur de droite Δ_d, et la taille de la liste tabou. Plusieurs stratégies ont été explorées et nous avons reporté celles réalisant un bon compromis entre la qualité des solutions obtenues et le temps d'exécution que nécessite l'algorithme. Nous avons donc retenu les valeurs suivantes :

1. Le critère d'arrêt pour les versions GH-LS et STS est fixé à un nombre maximum d'itérations égal à 500 pour les instances non corrélées (resp. 300 pour les instances corrélées).
2. La valeur du paramètre profondeur Δ est fixée à 5.
3. Pour la version MTS de l'algorithme, le nombre maximum d'itérations est fixé à 300 pour les instances non corrélées (resp. 200 pour les instances corrélées).
4. La profondeur de gauche Δ_g est fixée à 5 et celle de droite Δ_d est doublée (= 10).

Par ailleurs, pour les versions STS et MTS, la longueur de la liste mémoire varie dynamiquement. En effet, si m^* est le nombre des différentes classes, alors la longueur est arbitrairement et aléatoirement prise dans l'intervalle entier $[\lfloor\sqrt{m^*}\rfloor+1, \lfloor\sqrt{m^*}\rfloor+5]$. Le changement de la liste mémoire est opéré après 25 itérations sans amélioration obtenue de la meilleure solution courante. Ces limites permettent de garder un temps d'exécution moyen en dessous de 2 *minutes*. La qualité des solutions obtenues par les trois versions de l'algorithme est reportée dans le Tableau 3.2.

	GH-LS			*STS*			*MTS*		
Instances (#*nb*)	*Opt*	%Av. Rat.	Worst	*Opt*	%Av. Rat.	Worst	*Opt*	%Av. Rat.	Worst
Non corrélées (168)	18	0.210	1.455	81	0.051	0.476	129	0.019	0.316
Corrélées (72)	5	0.187	0.722	25	0.014	0.097	50	0.003	0.029

TAB. 3.2 – Représentation des résultats obtenus par les trois versions de l'algorithme.

En étudiant le Tableau 3.2, on remarque que :

1. Les résultats obtenus par l'algorithme GH-LS (voir les colonnes 2, 3 et 4 du Tableau 3.2) sont à 0.21% (pour les instances corrélées) et 0.187% (pour les insances non corrélées) de l'optimum. Cependant occasionnellement, on obtient des résultats de qualité inférieure pour certaines instances, avec un pourcentage au pire cas de 1.455%. Cela signifie que GH-LS donne seulement une première solution réalisable sans tenter de l'améliorer, puisque la liste mémoire a été supprimée du processus d'amélioration. L'historique des trajectoires des solutions (candidats) tabous n'est pas exploitée et de ce fait, nous pensons que le processus de recherche ne parvient pas à esquiver certains optima locaux.
2. En introduisant la liste tabou (STS), nous remarquons que l'algorithme GH-LS améliore sensiblement la qualité des solutions (voir les colonnes 5, 6 et 7 du Tableau 3.2). En effet la qualité moyenne des solutions est améliorée de 0.051% (resp. 0.014%) pour les instances non corrélées (resp. corrélées). Le nombre de solutions optimales obtenues augmente en passant de 18 à 81 (resp. de 5 à 25) pour les instances non corrélées (resp. corrélées). Globalement, le pourcentage des solutions optimales obtenues passe de 9.58% à 44.17%. Dans ce cas, la stratégie utilisant une liste mémoire permet d'améliorer plusieurs solutions et dans certains cas d'atteindre la solution optimale.

3. La version MTS de l'algorithme, utilisant deux paramètres de profondeur nécessite un temps d'exécution moyen plus important comparé aux deux autres versions (voir le Tableau 3.2 dans les colonnes 8, 9 et 10). Par contre, on obtient de meilleurs résultats même dans les cas les moins favorables (179 "bonnes" solutions obtenues sur 240 instances et le pourcentage de déviation de l'optimum est en moyenne de 0.011%). Ces résultats sont obtenus sans distinction pour les deux types d'instances avec un pourcentage de déviation en moyenne de 0.019% et 129 optima obtenus pour 168 instances non corrélées. Pour les instances corrélées, on obtient un pourcentage de déviation en moyenne de 0.003% et 50 solutions optimales pour 72 instances.

3.4.2 Discussion des tests numériques

Dans ce paragraphe, nous présentons une étude détaillée des tests numériques de la version MTS de l'algorithme. Comme nous l'avons reporté dans le Tableau 3.2, l'algorithme MTS, réalise de meilleures solutions même si ce dernier nécessite parfois un temps d'exécution plus important. Pour chaque instance, nous avons reporté (voir les Tableaux 3.3, 3.4 et 3.5) : le pourcentage de déviation (P.D.), le temps d'exécution T (mesuré en secondes) que l'algorithme MTS nécessite pour produire la solution finale. Nous avons également utilisé le symbole $\triangleright$ si l'algorithme obtient la solution optimale et le symbole $\diamond$ si MTS améliore la solution obtenue par l'algorithme STS. Les résultats pour les instances non corrélées sont reportés dans les Tableaux 3.3 et 3.4. Le Tableau 3.5 reporte les résultats obtenus pour les instances corrélées. A partir des Tableaux 3.3, 3.4 et 3.5, nous remarquons que :

1. Pour les instances non corrélées (voir les Tableaux 3.3 et 3.4), MTS obtient de meilleurs résultats que les résultats réalisés par STS (voir les colonnes sous $Meil$ marquées par le symbole $\diamond$). En effet, l'algorithme MTS améliore 75 solutions sur 168, ce qui représente un pourcentage de 44.64%. De plus, le temps d'exécution moyen que nécessite MTS est inférieur à 100 *secondes* pour les instances non corrélées.
2. Pour les instances corrélées (voir le Tableau 3.5), MTS améliore aussi des solutions produites par STS. Dans ce cas, l'algorithme MTS améliore 43 solutions sur 72, ce qui représente un pourcentage de 59.72% de solutions améliorées.

Algorithme MTS : cas non corrélé avec $m \leq 10$									
#Inst.	Opt.	Meil.	P.D.	T : temps d'exécution	#Inst.	Opt.	Meil.	P.D.	T : temps d'exécution
A02.1	20490	20490$^\diamond$	$\triangleright$	1.9	B02.1	50803	50803	$\triangleright$	6.2
A02.2	20419	20419	$\triangleright$	1.9	B02.2	50136	50136	$\triangleright$	5.5
A02.3	20889	20888	0.005	2.0	B02.3	50873	50873$^\diamond$	$\triangleright$	5.7
A02.4	20564	20564$^\diamond$	$\triangleright$	1.9	B02.4	52143	52143	$\triangleright$	6.0
A05.1	8071	8058$^\diamond$	0.161	3.3	B05.1	20371	20370$^\diamond$	0.005	5.3
A05.2	7995	7990$^\diamond$	0.063	2.8	B05.2	20349	20349$^\diamond$	$\triangleright$	5.6
A05.3	7960	7960$^\diamond$	$\triangleright$	3.3	B05.3	20424	20424$^\diamond$	$\triangleright$	5.8
A05.4	8115	8115$^\diamond$	$\triangleright$	2.6	B05.4	20376	20376$^\diamond$	$\triangleright$	5.9
A10.1	4054	4054	$\triangleright$	3.5	B10.1	10047	10043$^\diamond$	0.040	10.8
A10.2	3525	3525	$\triangleright$	2.7	B10.2	9905	9905$^\diamond$	$\triangleright$	8.1
A10.3	4087	4087$^\diamond$	$\triangleright$	3.1	B10.3	10129	10129$^\diamond$	$\triangleright$	7.4
A10.4	4037	4037$^\diamond$	$\triangleright$	5.3	B10.4	10360	10360$^\diamond$	$\triangleright$	9.1
C02.1	102284	102284	$\triangleright$	13.4	D02.1	153401	153401	$\triangleright$	23.5
C02.2	101565	101565	$\triangleright$	13.2	D02.2	152374	152374	$\triangleright$	23.8
C02.3	101551	101551	$\triangleright$	13.3	D02.3	153455	153455	$\triangleright$	24.5
C02.4	104568	104568$^\diamond$	$\triangleright$	15.2	D02.4	155556	155556	$\triangleright$	24.1
C05.1	40564	40564$^\diamond$	$\triangleright$	11.9	D05.1	61486	61480	0.010	19.0
C05.2	40798	40795$^\diamond$	0.007	12.8	D05.2	61000	61000$^\diamond$	$\triangleright$	19.1
C05.3	40438	40438$^\diamond$	$\triangleright$	10.7	D05.3	60690	60690$^\diamond$	$\triangleright$	19.4
C05.4	40449	40449$^\diamond$	$\triangleright$	11.4	D05.4	60750	60750	$\triangleright$	19.3
C10.1	20391	20391$^\diamond$	$\triangleright$	12.7	D10.1	30574	30569$^\diamond$	0.016	18.5
C10.2	20252	20252$^\diamond$	$\triangleright$	14.4	D10.2	30460	30460	$\triangleright$	24.5
C10.3	20213	20213$^\diamond$	$\triangleright$	15.8	D10.3	30619	30619$^\diamond$	$\triangleright$	19.6
C10.4	20858	20854$^\diamond$	0.019	14.5	D10.4	31029	31029$^\diamond$	$\triangleright$	19.8
E02.1	203577	203577	$\triangleright$	36.4	F02.1	409305	409305	$\triangleright$	98.8
E02.2	204750	204750	$\triangleright$	38.2	F02.2	409818	409818	$\triangleright$	99.7
E02.3	204462	204462	$\triangleright$	37.5	F02.3	409741	409741$^\diamond$	$\triangleright$	96.4
E02.4	207203	207203	$\triangleright$	36.7	F02.4	406213	406213$^\diamond$	$\triangleright$	91.6
E05.1	81275	81275$^\diamond$	$\triangleright$	27.6	F05.1	163514	163510$^\diamond$	0.002	65.5
E05.2	81240	81238$^\diamond$	0.002	29.4	F05.2	162566	162566	$\triangleright$	64.7
E05.3	81956	81956$^\diamond$	$\triangleright$	27.3	F05.3	163850	163850$^\diamond$	$\triangleright$	72.5
E05.4	81196	81194$^\diamond$	0.002	29.9	F05.4	163202	163202	$\triangleright$	53.6
E10.1	40681	40680$^\diamond$	0.002	24.5	F10.1	81739	81739	$\triangleright$	50.7
E10.2	40828	40828	$\triangleright$	26.7	F10.2	81957	81952$^\diamond$	0.006	52.4
E10.3	40839	40839$^\diamond$	$\triangleright$	24.0	F10.3	81917	81917$^\diamond$	$\triangleright$	54.0
E10.4	41406	41406	$\triangleright$	25.0	F10.4	81168	81168	$\triangleright$	41.7

TAB. 3.3 – Performance de l'approche recherche à profondeur multiple (MTS) sur les instances non corrélées avec $m \leq 10$ (classes).

Algorithme MTS : cas non corrélé avec $m \geq 20$									
#Inst.	Opt.	Meil.	P.D.	T :temps d'exécution	#Inst.	Opt.	Meil.	P.D.	T : temps d'exécution
A20.1	1989	1987$^{\diamond}$	0.100	8.6	A30.1	1088	1088	▷	16.4
A20.2	1465	1465	▷	9.8	A30.2	747	747	▷	20.8
A20.3	2001	2001$^{\diamond}$	▷	4.4	A30.3	1277	1277	▷	10.8
A20.4	1972	1972$^{\diamond}$	▷	8.1	A30.4	1198	1198	▷	10.7
A40.1	712	712	▷	10.7	A50.1	550	550	▷	55.9
A40.2	595	595	▷	37.8	A50.2	459	459	▷	43.8
A40.3	716	716	▷	31.6	A50.3	536	536	▷	52.9
A40.4	668	668	▷	36.0	A50.4	489	489	▷	59.8
B20.1	4997	4991$^{\diamond}$	0.120	12.3	B30.1	2525	2525	▷	22.3
B20.2	4164	4164	▷	10.3	B30.2	3123	3123	▷	12.2
B20.3	4996	4981	0.300	2.7	B30.3	3218	3218	▷	22.5
B20.4	5115	5115$^{\diamond}$	▷	13.0	B30.4	2716	2716	▷	12.1
B40.1	2173	2173	▷	30.7	B50.1	1811	1811	▷	58.8
B40.2	2177	2177	▷	40.3	B50.2	1615	1615	▷	40.7
B40.3	2181	2181	▷	20.4	B50.3	1511	1511	▷	62.5
B40.4	2057	2057	▷	39.5	B50.4	1780	1780	▷	40.9
C20.1	10113	10113	▷	17.3	C30.1	6719	6698	0.313	26.6
C20.2	10066	10047	0.189	25.1	C30.2	6698	6690$^{\diamond}$	0.119	31.2
C20.3	10079	10065	0.139	17.4	C30.3	6594	6594$^{\diamond}$	▷	48.0
C20.4	10377	10377$^{\diamond}$	▷	20.5	C30.4	6206	6206	▷	12.5
C40.1	4644	4644	▷	24.5	C50.1	3935	3935	▷	28.6
C40.2	4846	4846	▷	41.6	C50.2	3992	3980$^{\diamond}$	0.301	0.6
C40.3	4588	4588	▷	33.3	C50.3	3633	3633	▷	54.0
C40.4	5122	5122$^{\diamond}$	▷	34.9	C50.4	3994	3994	▷	48.6
D20.1	15276	15272$^{\diamond}$	0.026	23.4	D30.1	10129	10115$^{\diamond}$	0.138	33.6
D20.2	15151	15151	▷	27.8	D30.2	10103	10103$^{\diamond}$	▷	45.1
D20.3	15256	15253$^{\diamond}$	0.020	25.7	D30.3	10153	10146$^{\diamond}$	0.069	29.8
D20.4	15468	15447	0.136	27.6	D30.4	9591	9591	▷	12.8
D40.1	7606	7582	0.316	38.3	D50.1	6057	6057	▷	1.0
D40.2	7505	7505$^{\diamond}$	▷	43.0	D50.2	5797	5797	▷	62.0
D40.3	7074	7074	▷	22.9	D50.3	5407	5407	▷	54.6
D40.4	7663	7652$^{\diamond}$	0.144	45.6	D50.4	6131	6131$^{\diamond}$	▷	0.9
E20.1	20274	20274	▷	31.6	E30.1	13438	13422	0.119	56.2
E20.2	20382	20382	▷	33.7	E30.2	13556	13556$^{\diamond}$	▷	34.2
E20.3	20368	20358$^{\diamond}$	0.049	28.0	E30.3	13590	13590$^{\diamond}$	▷	53.0
E20.4	20634	20634$^{\diamond}$	▷	38.8	E30.4	13200	13200	▷	13.5
E40.1	10098	10098	▷	49.4	E50.1	8081	8079$^{\diamond}$	0.025	1.4
E40.2	9838	9838	▷	34.3	E50.2	8081	8079	0.025	1.4
E40.3	10150	10150$^{\diamond}$	▷	64.2	E50.3	8111	8111	▷	1.4
E40.4	10260	10248$^{\diamond}$	0.117	48.4	E50.4	8195	8195	▷	1.5
F20.1	40884	40884	▷	69.6	F30.1	27217	27217	▷	63.7
F20.2	40926	40926$^{\diamond}$	▷	67.3	F30.2	27250	27244	0.022	61.0
F20.3	40936	40936$^{\diamond}$	▷	58.6	F30.3	27223	27223$^{\diamond}$	▷	62.2
F20.4	40528	40528	▷	52.5	F30.4	26938	26938	▷	79.8
F40.1	20393	20390$^{\diamond}$	0.015	83.9	F50.1	16262	16259$^{\diamond}$	0.018	4.4
F40.2	20425	20425	▷	97.0	F50.2	16291	16288$^{\diamond}$	0.018	3.5
F40.3	20428	20428$^{\diamond}$	▷	70.0	F50.3	16326	16326	▷	3.5
F40.4	20218	20218	▷	73.0	F50.4	16123	16114$^{\diamond}$	0.056	79.7

Tab. 3.4 – Performance de l'approche recherche à profondeur multiple (MTS) sur les instances non corrélées avec $m \geq 20$ (classes).

Algorithme MTS : cas corrélé									
#Inst.	Opt.	Sol.	P.D.	T :temps d'exécution	#Inst.	Opt.	Sol.	P.D.	T :temps d'exécution
A02C.1	41492	41492$^\diamond$	$\triangleright$	3.0	D02C.1	312223	312217	0.002	14.6
A02C.2	41397	41397	$\triangleright$	1.6	D02C.2	311947	311947	$\triangleright$	14.6
A02C.3	41565	41565	$\triangleright$	2.0	D02C.3	311690	311690$^\diamond$	$\triangleright$	14.7
A02C.4	41556	41556	$\triangleright$	1.6	D02C.4	311419	311417$^\diamond$	0.001	14.5
A05C.1	16554	16554	$\triangleright$	1.9	D05C.1	124794	124782	0.010	12.1
A05C.2	16561	16561$^\diamond$	$\triangleright$	1.7	D05C.2	124758	124758	$\triangleright$	11.9
A05C.3	16590	16587	0.018	3.1	D05C.3	124669	124669	$\triangleright$	11.9
A05C.4	16584	16584	$\triangleright$	3.9	D05C.4	124529	124529$^\diamond$	$\triangleright$	11.6
A10C.1	8245	8245	$\triangleright$	2.9	D10C.1	62334	62334$^\diamond$	$\triangleright$	9.6
A10C.2	8203	8203$^\diamond$	$\triangleright$	4.3	D10C.2	62311	62311$^\diamond$	$\triangleright$	10.0
A10C.3	8250	8248$^\diamond$	0.024	4.6	D10C.3	62289	62280$^\diamond$	0.014	9.8
A10C.4	8255	8255	$\triangleright$	5.6	D10C.4	62216	62211$^\diamond$	0.008	10.3
B02C.1	103672	103672$^\diamond$	$\triangleright$	2.8	E02C.1	416396	416396$^\diamond$	$\triangleright$	23.4
B02C.2	103572	103555$^\diamond$	0.016	3.2	E02C.2	415426	415426$^\diamond$	$\triangleright$	24.2
B02C.3	104034	104034	$\triangleright$	2.9	E02C.3	415470	415470	$\triangleright$	24.1
B02C.4	104031	104031	$\triangleright$	2.7	E02C.4	415341	415334$^\diamond$	0.002	23.6
B05C.1	41393	41393$^\diamond$	$\triangleright$	2.7	E05C.1	166484	166474$^\diamond$	0.006	14.1
B05C.2	41437	41425$^\diamond$	0.029	3.1	E05C.2	166126	166126$^\diamond$	$\triangleright$	14.2
B05C.3	41580	41580$^\diamond$	$\triangleright$	2.7	E05C.3	166116	166094	0.013	14.5
B05C.4	41561	41561	$\triangleright$	3.7	E05C.4	166062	166062	$\triangleright$	13.7
B10C.1	20657	20657	$\triangleright$	6.3	E10C.1	83216	83199$^\diamond$	0.020	12.4
B10C.2	20648	20648	$\triangleright$	7.2	E10C.2	82995	82985$^\diamond$	0.012	15.5
B10C.3	20740	20740$^\diamond$	$\triangleright$	6.8	E10C.3	83029	83014$^\diamond$	0.018	14.7
B10C.4	20721	20721$^\diamond$	$\triangleright$	7.7	E10C.4	82981	82981$^\diamond$	$\triangleright$	15.7
C02C.1	207675	207673$^\diamond$	0.001	7.6	F02C.1	830622	830611$^\diamond$	0.001	82.9
C02C.2	207916	207916	$\triangleright$	8.1	F02C.2	831513	831513	$\triangleright$	82.9
C02C.3	208043	208043	$\triangleright$	10.1	F02C.3	831145	831145$^\diamond$	$\triangleright$	83.9
C02C.4	207973	207973$^\diamond$	$\triangleright$	10.8	F02C.4	831445	831445$^\diamond$	$\triangleright$	85.4
C05C.1	83013	83013$^\diamond$	$\triangleright$	6.2	F05C.1	332143	332126	0.005	41.6
C05C.2	83129	83120$^\diamond$	0.011	6.5	F05C.2	332527	332527	$\triangleright$	41.9
C05C.3	83185	83185	$\triangleright$	6.6	F05C.3	332402	332388	0.004	42.5
C05C.4	83188	83188	$\triangleright$	5.9	F05C.4	332498	332498$^\diamond$	$\triangleright$	40.8
C10C.1	41466	41466$^\diamond$	$\triangleright$	7.7	F10C.1	166028	166028$^\diamond$	$\triangleright$	29.9
C10C.2	41521	41518$^\diamond$	0.007	10.7	F10C.2	166248	166248$^\diamond$	$\triangleright$	28.6
C10C.3	41554	41554	$\triangleright$	12.0	F10C.3	166154	166148	0.004	29.8
C10C.4	41554	41554$^\diamond$	$\triangleright$	6.5	F10C.4	166192	166192	$\triangleright$	30.4

Tab. 3.5 – Performance de l'approche recherche à profondeur multiple (MTS) sur les instances corrélées.

3.5 Conclusion

Dans ce chapitre, nous nous sommes intéressés à la résolution approchée du problème de la distribution équitable. Dans un premier temps, nous avons proposé une première version de l'algorithme exploitant certaines caractéristiques de la recherche tabou. Le principe de cette version de l'algorithme consiste : (i) à utiliser un paramètre profondeur limitant la recherche des solutions voisines autour des éléments dits "critiques" et (ii) à introduire une mémoire permettant la non stagnation de certaines solutions. Dans un deuxième temps, nous avons proposé une deuxième version de l'algorithme. L'idée principale consiste à tenter la combinaison entre l'intensification de la recherche dans l'espace des solutions et la diversification de la solution. La stratégie adaptée portait sur la modification de la façon d'exploiter le paramètre profondeur. Les deux versions de l'algorithme ont été testées sur un nombre d'instances de la littérature. Nous avons souligné la rapidité de la première version et l'efficacité de la deuxième. En effet, même si les deux versions de l'algorithme sont en $O(mn)$, l'étude expérimentale montrait que, par rapport à la première version de l'algorithme, la deuxième version produisait un nombre plus important de solutions optimales mais nécessitait un temps d'exécution plus important. Cette différence est expliquée par le fait que dans le calcul de la complexité des deux approches, les paramètres profondeur simple Δ et profondeur double (Δ_g, Δ_d) sont considérés comme des constantes. Cependant, dans la pratique, le temps de calcul pour explorer l'espacede recherche est plus important lorsqu'on utilise la stratégie en double profondeur.

Chapitre 4

Méthodes de recherche locale pour le sac-à-dos généralisé à choix multiple

Dans ce chapitre, nous proposons deux méthodes de résolution approchée. Une première méthode qui s'appuie sur une "recherche guidée" qui est par la suite combinée avec une stratégie de pénalisation sur des composantes de la fonction objectif. Une deuxième méthode qui s'appuie principalement sur la recherche locale. ([1], [2])

4.1 Introduction

Ce chapitre est consacré à l'étude du problème de sac-à-dos généralisé à choix multiple. Nous nous intéressons principalement à la résolution approchée. Nous allons développer deux méthodes de recherche locale itérative. Ces approches sont connues dans la littérature sous le nom de *iterative local search method* (voir Kilby *et al.* [29], Voudouris *et al.* [59, 60], et Tsang *et al.* [57]).

---

[1] Ce chapitre fait l'objet d'un document de travail que nous avons publié sous le titre : *"Heuristic Algorithms for the Multiple-Choice Multidimensional Knapsack Problem"*, dans les Cahiers de la MSE, 2003.31 et accepté pour publication dans Journal of the Operational Research Society.

[2] Ce chapitre fait l'objet d'un document de travail que nous avons publié sous le titre : *"A Reactive Local Search-Based Algorithm for the Multiple-Choice Multidimensional Knapasck Problem"*, dans les Cahiers de la MSE, 2003.33 et soumis à Computational Optimization and Applications (2003).

La première approche est vue comme une méthode de recherche guidée proprement dite avec une notion de pénalité sur la fonction objectif. Cette dernière s'appuie sur deux phases :

1. La première phase est une phase de pénalisation d'une solution non améliorée et stagnée autour d'un optimum local.
2. La seconde phase est appliquée (i) pour normaliser la configuration de la solution pénalisée et (ii) pour tenter de l'améliorer.

La deuxième approche est une hybridation entre la recherche tabou et d'autres stratégies de recherche locale. Dans notre cas, nous l'appelons une recherche locale réactive où la répétition de certaines configurations (solutions) est mise *tabou* au cours de la recherche. L'algorithme tente de démarrer par une solution réalisable, ensuite plusieurs stratégies sont appliquées :

1. Les stratégies *débloquer* et *dégrader* une solution courante sont utilisées pour :
   - (*i*) tenter d'échapper à un optimum local qui peut figer la recherche,
   - (*ii*) introduire une diversification de la recherche.
2. Une liste dite *mémoire* est introduite pour interdire toute répétition dans les configurations visitées auparavant.

4.2 Méthodes de recherche locale

Dans cette section, nous allons décrire l'idée principale de la recherche locale. En général, une recherche locale peut se résumer de la façon suivante :

(i) partir d'une solution ;

(ii) améliorer la solution ;

(iii) répéter le processus un certain nombre de fois.

Il n'est malheureusement pas possible pour tout problème de définir des modifications raisonnables de sorte que, à partir de n'importe quelle solution, il soit toujours possible d'obtenir une solution optimale ou même une bonne solution approchée en n'utilisant que des modifications améliorantes. Par conséquent, avec un tel schéma, il est nécessaire de pouvoir détériorer une solution, même s'il existe parfois des modifications améliorantes, dans l'espoir d'arriver plutard à une meilleure solution. Pourqu'une

recherche locale itérative puisse déterminer quelles modifications détériorantes doivent être réalisées, on doit lui assigner des règles qui lui permettent de se diriger dans l'espace des solutions.
Un des buts du présent chapitre est précisément de dégager certaines règles permettant une meilleure exploration des solutions. En outre, nous pensons que ces règles, dans un cadre général, dépendent fortement du problème que l'on cherche à résoudre. Pour ce qui concerne le problème MMKP, ces règles seront explicitées ultérieurement. Avant tout nous rappelons que le problème du MMKP est donné par :

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

où la variable x_{ij} vaut 0, si l'objet j de la i-ème classe n'est pas choisi et vaut 1 sinon. Sans perte de généralité et pour éviter les solutions triviales, nous considérons que :

$$\forall k \in \{1,\dots,m\} \ \sum_{i=1}^{n} \min_{1\leq j\leq r_i}\left\{w_{ij}^k\right\} \leq C^k \leq \sum_{i=1}^{n} \max_{1\leq j\leq r_i}\left\{w_{ij}^k\right\}. \tag{4.1}$$

4.3 Représentation de la solution

Avant de décrire les deux approches, nous donnons une représentation appropriée d'une solution du MMKP ainsi que des notations qui vont nous servir par la suite.
Soit $\mathcal{J}$ un ensemble d'objets répartis sur n classes disjointes telles que j, $j \in J_i$, possède la valeur profit v_{ij} et un vecteur poids $W_{ij} = (w_{ij}^1,\dots,w_{ij}^m)$ et, soit $C = (C^1,\dots,C^m)$ un vecteur capacité pour le sac-à-dos multidimensionnel. Ce sac-à-dos est soumis à des contraintes à choix multiple qui peuvent être formulées de la manière suivante :

$$\forall i \in \{1,\dots,n\},\ \sum_{j=1}^{r_i} x_{ij} = 1.$$

L'objectif est de déterminer un sous-ensemble d'objets de valeur maximale et pouvant être inclus dans le sac sans violer les contraintes. La représentation standard d'une

solution binaire du MMKP est triviale puisqu'elle décrit les variables positives du problème. La figure 4.1 représente un vecteur solution.

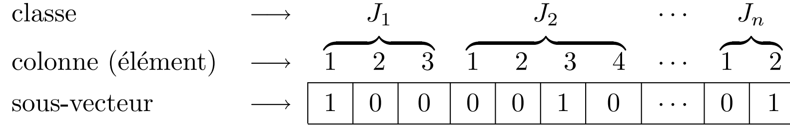

Fig. 4.1 – Représentation d'une solution de MMKP.

On dit qu'une solution est réalisable si :

$$\forall k \in \{1, \ldots, m\}, \ \sum_{i=1}^{n} \sum_{j=1}^{r_i} w_{ij}^k x_{ij} \leq C^k.$$

Et pour chaque classe J_i, on choist *un et un seul* objet j, c'est-à-dire, $x_{ij} = 1$ si l'objet j de la classe J_i est sélectionné et $x_{ij} = 0$, sinon.
Dans la suite, nous distinguerons deux types d'états : *état réalisable* (ER) et *état non réalisable* (EN). Ces états sont représentés par les inégalités suivantes :

$$(ER) : \ \forall k \in \{1, \ldots, m\}, \ \sum_{i=1}^{n} \sum_{j=1}^{r_i} w_{ij}^k x_{ij} \leq C^k.$$

$$(EN) : \ \exists k \in \{1, \ldots, m\}, \ \sum_{i=1}^{n} \sum_{j=1}^{r_i} w_{ij}^k x_{ij} > C^k.$$

ER signifie que la solution en cours, (notée S), ne viole pas les contraintes de capacité, et EN signifie qu'il existe au moins une contrainte de capacité violée par S. L'objectif est de construire des états ER améliorés (ou de transformer tout état EN en ER) en appliquant dans la recherche une stratégie d'*inter-change local*.

4.3.1 Une solution initiale

La solution initiale est obtenue en appliquant une *procédure constructive*, notée CP. CP est une procédure gloutonne à deux phases : une phase REJET et une autre phase AJOUT. Ceci est pour obtenir une solution en considérant le procédé générant un état ER. Elle commence par calculer le rapport pseudo-utilité $u_{ij} = \frac{v_{ij}}{\langle C, W_{ij} \rangle}, \ j \in \{1, \ldots, r_i\}$,

où $\langle ., . \rangle$ est le produit scalaire dans $I\!R^m$. Ensuite, elle sélectionne l'élément j de chaque classe $J_i,\ i \in \{1, \ldots, n\}$, qui réalise le plus grand rapport u_{ij}. Si la solution obtenue est d'état FS, alors CP s'arrête, sinon (phase REJET), elle considère la contrainte la plus violée parmi toutes les contraintes et qu'on note C^{k_0}. Relativement à cette contrainte C^{k_0}, elle sélectionne la classe J_{i_0} correspondant à l'élément fixé j_{i_0} ayant le plus grand poids $w^{k_0}_{i_0 j_{i_0}}$ par rapport à tous les éléments fixés et relativement à cette contrainte la plus violée C^{k_0}. Cet élément (phase AJOUT) est alors échangé avec un autre élément j sélectionné dans la même classe J_{i_0}, et la procédure contrôle la réalisabilité de l'état. Si la nouvelle solution obtenue est d'état EN, alors elle sélectionne l'élément j'_{i_0} de plus faible poids dans la classe déjà sélectionnée J_{i_0}, qui à son tour devient le nouvel élémnent fixé dans cette classe. Ce processus est réitéré jusqu'à obtention d'un état ER ou du plus faible taux d'irréalisabilité.

On peut voir que la procédure CP a une complexité au pire égale à $O(max\{\theta m, n\})$, où $\theta = \max\{r_1, \ldots, r_n\}$. En effet, à l'étape générale (la boucle `tant que`), la procédure utilise m opérations pour obtenir k_0 et n opérations pour obtenir i_0. Par conséquent, la procédure `NouvelleConfig()`, utilise $m\theta$ opérations pour construire une nouvelle configurations, tandis que le nombre d'opérations consommées par `MettreàJour()` est borné au pire par $(m + 1)\theta$ opérations. D'où la complexité au pire de CP est évaluée à $O(m\theta + n + \theta)$, qui est équivalente à $O(m\theta + n)$ ce qui est équivalent aussi à $O(max\{m\theta, n\})$.
Dans ce qui suit, nous décrivons un algorithme complémentaire de recherche locale dans le but d'améliorer la qualité de la solution générée par CP.
Les principales étapes de l'approche CP sont décrites dans la figure 4.2 :

```
Entrée : Une instance de MMKP.
Sortie : Une solution S avec sa valeur O(S).

Initialisation.
 Pour i = 1,...,n, mettre u_{ij_i} = max{u_{ij}, j = 1,...,r_i} ;
     S_i ⟵ j_i ;
     Poser φ[i] = j_i ; x_{iφ[i]} = 1 ;
     Poser R^k = ∑_{i=1}^n w^k_{iφ[i]}, ∀k = 1,...,m ;
          /* R^k : les poids cumulés pour la contrainte k */
 FinPour ;
 S = (S_1,...,S_n) ;
Étape générale.
 Tant que (R^k > C^k, for k = 1,...,m) /* phase REJET */
     k_0 ⟵ argmax_{1≤k≤m} {R^k − C^k};
     i_0 ⟵ argmax_{1≤i≤n} {w^{k_0}_{ij_i}};
     φ[i_0] = j_{i_0} ; x_{i_0 φ[i_0]} = 0 ;
     R^k = R^k − w^k_{i_0 φ[i_0]} for k = 1,...,m ;
     Pour j = 1,...,r_{i_0} /* phase AJOUT */
       Si (∃j ≠ j_{i_0} and R^k + w^k_{i_0 j} < C^k, for k = 1,...,m) alors
         x_{i_0 j} = 1 ;
         j_{i_0} = j ;
         φ[i_0]=j_{i_0} ;
         R^k = R^k + w^k_{i_0 φ[i_0]}, pour k = 1,...,m ;
         S=(φ[i_0]; φ[i], ∀i ≠ i_0, i = 1,...,n) est une solution réalisable ;
         Sortir ave le vecteur S ;
       FinSi ;
     FinPour ;
     j'_{i_0} ⟵ arg min_{1≤j≤r_{i_0}} {w^{k_0}_{i_0 j}}; /* si la solution obtenue n'est pas réalisable */
     j_{i_0} = j' ; φ[i_0] = j_{i_0} ; x_{i_0 φ[i_0]} = 1 ;
 FinTant_Que ;
 Retourner S avec la valeur O(S).
```

FIG. 4.2 – La procédure constructive : CP.

4.3.2 Discussion de la non réalisablité d'une solution

Nous traitons dans ce paragraphe le cas de la non réalisablité des solutions obtenues par CP c'est-à-dire celles d'état EN. Dans ce cas, nous essayons de réduire la non réalisabilité de la solution obtenue en faisant appel à une procédure similaire à CCP. Cette procédure s'appuie sur une stratégie d'interchange local entre deux éléments d'une même classe. Le meilleur interchange opéré pour un élément j_i d'une classe i est

celui qui satisfait un critère de décision ([3]) qui réalise le minimum du rapport suivant :

$$D_{j_i} = \frac{\sum_{k=1}^{m} R^k}{\sum_{k=1}^{m} C^k}.$$

Cet interchange se fait en deux étapes :

1. Calcul des rapports D_{j_i}
   Dans un premier temps, nous effectuons l'interchange réalisant le meilleur critère de décision que nous avons défini précedemment. Plus précisément, pour chaque classe i, $i = 1, \dots, n$: (a) nous effectuons un interchange entre deux éléments, (b) pour chaque élément j_i interchangé, nous calculons le rapport D_{j_i}, (c) ensuite nous sauvegardons le plus petit rapport $D_{j_{0i}}$ ainsi que l'indice j_{0i} de l'élément correspondant à ce rapport :
2. Sélection du meilleur rapport
   A partir des rapports correspondant à chaque classe et stockés précedemment, nous sélectionnons l'indice i_0 de la classe qui réalise le meilleur (plus petit) rapport. Ensuite, nous effectuons l'interchange dans la classe i_0. Nous terminons cette étape par la mise à jour des ressources consommées.

On relance ce processus pour plusieurs itérations définies par $MaxIter$, et à chaque itération on vérifie l'état de réalisabilité de la solution. Dans le cas où la solution est d'état ER, le processus s'arrête et la solution obtenue est réalisable.
Dès que le processus atteint le nombre maximal d'itérations sans obtenir une solution réalisable, on dit que le processus est incapable de générer une solution réalisable. Également, nous pouvons appliquer les procédures que nous allons développer dans la suite pour tenter de transformer un état non réalisable en un état réalisable.

Dans ce qui suit, nous décrivons une procédure de recherche locale complémentaire que nous notons CCP dans le but d'améliorer la qualité de la solution générée par la procédure CP.

4.3.3 Une recherche locale complémentaire

La recherche complémentaire que nous proposons est une recherche locale itérative. Elle repose sur l'idée d'amélioration itérative d'une solution donnée. Elle applique donc :

[3]On peut evidemment choisir d'autres critères de décision

(i) des stratégies d'interchange (*swapping*) d'éléments considérés comme des *anciens éléments* ;

(ii) et une phase de changement de fixation d'éléments qui consiste à remplacer l'ancien élément par un nouvel élément sélectionné dans la même classe.

On dit que l'échange est autorisé s'il permet d'obtenir un état ER, sinon si l'état est EN, il passe à un autre interchange jusqu'à obtention d'un échange autorisé. Par ce procédé, l'interchange sera généralisé aux éléments restants de la même classe. Ceci est pour sélectionner un nouvel élément réalisant la meilleure valeur de solution locale pour la classe courante. Ensuite, les deux éléments, notés j_i et j_i', appartenant à la même classe, notée J_i, sont échangés dans la nouvelle solution qui réalise la meilleure évaluation de toutes les classes. Ce procédé est réitéré jusqu'à ce qu'un critère d'arrêt soit atteint.

Nous décrivons les principales étapes de cet algorithme, que nous avons noté CCP, dans la figure 4.3.

Tout d'abord, nous détaillons le principe de la procédure `RechercheSwapLocal()` utilisée par CCP et qui permet d'améliorer la solution de départ donnée par CP.

A) Au départ, la procédure `RechercheSwapLocal()` initialise le meilleur élément à interchanger (*swapper*) :

- **A.1)** *valeur* $\longleftarrow v_{iS_i}$, où v_{iS_i} représente le profit de l'ancien élément fixé dans la i-ème classe J_i à interchanger ;
- **A.2)** $s_i \longleftarrow S_i$, où s_i est un élément candidat dans J_i à interchanger ;

B) Ensuite, elle opère l'échange qui a été autorisé :

- **B.1)** `Pour` $(j = 1, \ldots, r_i$ `et` $j \neq S_i)$ `faire`
  - `Si` $(v_{ij} > \mathit{valeur}$ `et` $R^k - w^k_{iS_i} + w^k_{ij} \leq C^k,\ \forall k = 1, \ldots, m)$ `alors`
    - Set *valeur* $\longleftarrow v_{ij}$;
    - Set $s_i \longleftarrow j$;
- **B.2)** `Retourner` s_i comme étant le meilleur élément à interchanger localement.

Nous rappelons d'une part que CP a une complexite de $O(max\{m\theta, n\})$. D'autre part, CCP utilise la procédure CP pour démarrer avec une solution initiale (`Eatpe 1`). A l'étape générale (`Etape 2`), la procédure `RechercheSwapLocal`(), requiert $2m$ opérations pour mettre à jour les ressources consommées et une opération pour calculer la valeur $O(S)$. Par ailleurs, l'algorithme consomme $\theta(2m+1)$ opérations pour fixer une classe et $\theta(2m+1) \times n$ opérations pour fixer toutes les classes. Par conséquent, on peut

```
Entrée : Une solution réalisabe S avec sa valeur O(S).
Sortie : Une solution réalisable améliorée S* avec sa valeur O(S*).

Eatpe 1.
  mettre S = (S_1, ..., S_n) ⟵ CP();
  mettre S* ⟵ S;
Etape 2.
  Tant que non Condition_d'Arrêt() faire
    pour tout i dans {1, ..., n} faire
      j'_i ⟵ RechercheSwapLocal(j_i, J_i)
          /* j_i (j'_i) représente l'ancien (nouvel) élément de la classe J_i */
          /* et l'interchange entre j_i and j'_i est autorisé */
      S_i ⟵ j'_i ;
      S ⟵ (S_1, ..., j'_i, ..., S_n) ;
      Si O(S_1, ..., j'_i, ..., S_n) > O(S*) alors S* ⟵ (S_1, ..., j'_i, ..., S_n);
      FinSi
  FinTant_Que
  retourner S* avec sa valeur O(S*);
```

FIG. 4.3 – La procédure complémentaire CCP.

borner la compléxité au pire de CCP par $MaxIter \times O(\theta(2m+1) \times n + (m+\ n) + \theta(2m+1))$ ce qui est équivalent à dire que la compléxité de CCP est évaluée au pire cas à $O(\theta(m+n))$, qui est équivalente à $O(\theta max\{m, n\})$.

Partie A : la recherche locale guidée

4.4 La recherche locale guidée

La recherche locale guidée, GLS (*Guided Local Search*), est l'une des approches récentes considérées comme une métaheuristique de résolution des problèmes d'optimisation combinatoire. Cette approche a été utilisée pour la première fois par Voudouris et Tsang [59] pour résoudre des problèmes de satisfaction de contraintes. Elle peut être vue comme une méthode tabou (voir Hansen [16] et Glover et Laguna [14]) puisqu'elle s'appuie sur la notion de mémoire qui contrôle le processus de recherche d'une façon similaire à celle opérée par la méthode tabou.

GLS est une méthode générale d'optimisation pour résoudre des problèmes d'optimisation combinatoire tels que le problème du voygeur de commerce (Voudouris et Tsang [61]) ou le problème de la programmation par contraintes (Voudouris et Tsang [59]) ou le problème de tournées de véhicules (voir Kilby *et al.*[29]). La méthode GLS s'appuie sur l'idée de guider la recherche locale dans l'espace de recherche. En effet, cette approche applique principalement la modification de la fonction objectif par l'ajout d'une pénalité ou d'un ensemble de termes de pénalité. La recherche locale se trouve donc orientée par les termes de pénalité et concentre la recherche sur des régions différentes de l'espace de solutions. Le processus est réitéré en procédant à des appels itératifs de la recherche locale.
Si durant le processus de recherche, cette dernière est piégée autour d'un optimum local, alors un changement des termes de pénalité est appliquée, puis la recherche est de nouveau relancée. La modification des termes de pénalité permet de régulariser la solution générée par la recherche pourqu'elle corresponde aux informations initiales du problème.
L'idée de l'approche est similaire à une large classe d'algorithmes d'optimisation combinatoire telles que la méthode tabou. En fait, GLS peut être considérée comme une méthode tabou avec des différences dans le concept. La méthode tabou est caractérisée par les interdictions des mouvements qui sont généralement implémentées sous forme de listes (listes tabous). Contrairement à GLS, ces interdictions se réfèrent uniquement aux solutions et sous forme de pénalisation de la fonction objectif. En plus, dans la méthode tabou ces interdictions peuvent être très restrictives et uniquement relachées par un critère d'aspiration.

En général, GLS effectue des transitions d'un optimum local en pénalisant des

configurations de solutions ne figurant pas dans la solution approchée. Elle définit une fonction objectif modifiée, augmentée par un ensemble de paramètres de pénalité sur ces configurations. La fréquence de la recherche locale et la mise à jour des paramètres de pénalisation peuvent être réitérés aussi souvent que celà est nécessaire. La figure 4.4 résume le principe de l'approche GLS.

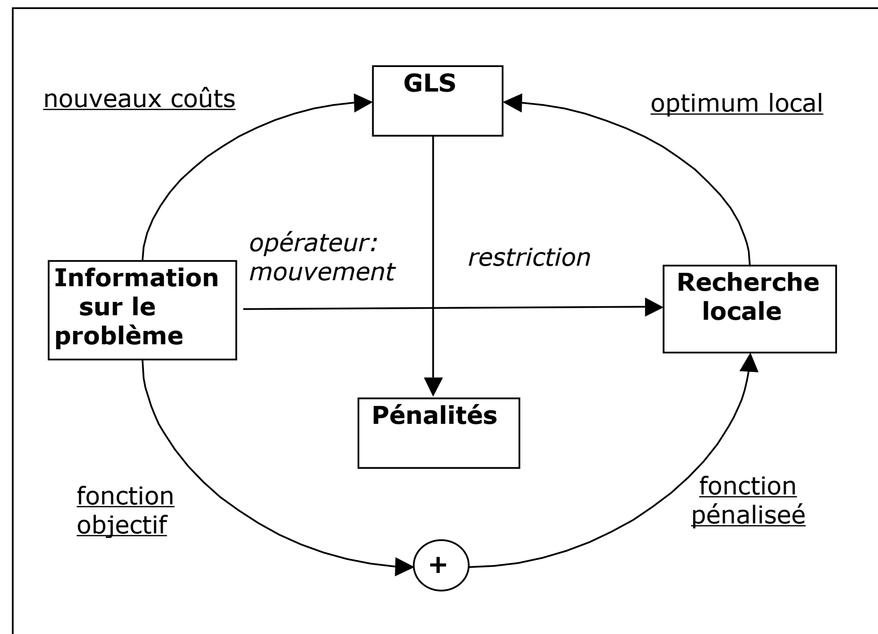

FIG. 4.4 – L'approche GLS.

4.5 Adaptation de la recherche locale guidée

Dans notre étude, nous proposons une variante de la méthode de recherche guidée qui consiste en une pénalisation du processus de recherche pour éviter aux solutions de stagner autour des optimas locaux. La pénalisation opérée est contrôlée par deux paramètres. Un premier paramètre nommé *exploration* qui applique des pénalisations sur les valeurs profits d'un certain nombre d'éléments repérés pendant la recherche, et un deuxième paramètre appelé *diamètre* qui cible l'étendue de l'espace de recherche. Cette approche utilise également des coefficients de pénalisation choisis par rapport à la taille

et la nature du problème traité afin de pénaliser aléatoirement toute solution semblant indiquer être piégée autour d'un optimum local. L'objectif de cette pénalisation est double :

- (i) libérer la solution courante de toute zone attractive estimée non améliorante.
- (ii) changer de trajectoire de recherche.

Cette dernière stratégie est introduite principalement pour diversifier la recherche et explorer d'autres régions.

4.6 Un algorithme utilisant la pénalisation et la transformation normalisante

Dans ce paragraphe, nous décrivons le principe général de l'approche recherche guidée appliquée au problème MMKP. Celle-ci applique :

1. Une stratégie de pénalisation.
2. Une stratégie de normalisation des configurations des solutions retenues.

L'algorithme démarre par une solution réalisable, notée S^* (obtenue en appliquant les procédures constructives CP et CCP développées dans les sections 4.2 et 4.3). La dernière solution obtenue est considérée comme la meilleure solution réalisable trouvée, sans application de la stratégie de pénalisation.

4.6.1 Le principe général de l'algorithme

L'algorithme proposé peut être considéré comme un alorithme utilisant deux phases : (i) phase d'application d'une phase de pénalisation, et (ii) phase de normalisation. Cette dernière phase permet de revenir aux configurations des solutions du problème original.

- D'une part, la phase de pénalisation est appliquée si on ne peut améliorer la solution courante après qu'un certain nombre d'itérations soit effectué par l'algorithme. Dans ce cas, un facteur de pénalisation est appliqué pour transformer les valeurs profits de la fonction objectif. Partant de la nouvelle configuration obtenue (qui demeure une solution réalisable pour le poblème original), le procédé

consiste à trouver un meilleur voisinage pour améliorer localement la configuration courante.
– D'autre part, on applique la phase de normalisation qui a pour objectif de retransformer la dernière configuration pénalisée en une solution réalisable pour le MMKP. La solution issue de ce procédé est normalisée car les valeurs profits de sa fonction objectif redeviennent les valeurs profits du problème original.

4.6.2 Description de l'algorithme DerAlg

Les principales étapes de l'algorithme sont décrites dans la figure 4.5. L'algorithme démarre en appliquant la procédure constructive CP pour tenter de produire une solution initiale réalisable. La configuration de la solution courante est stockée dans un vecteur ρ. La meilleure solution (resp. évaluation) S (resp. $O(S)$) est initialement affectée à la solution (resp. évaluation) initiale S^* (resp. $O(S^*)$). A l'étape générale, on applique `phase normale` qui correspond à la phase sans pénalisation. Celle-ci applique une *recherche d'interchange local* dont l'objectif est d'accroître la valeur de la première solution réalisable obtenue. A chaque itération de l'étape générale, la meilleure solution courante est mise à jour si la solution en cours est améliorée. Dans ce cas, on distingue deux cas :

– d'une part, l'étape est mise en `phase normale` et la solution est celle fixée à S^* et de valeur $O(S^*)$,
– d'autre part, si l'étape est mise en `phase de pénalisation`, alors la procédure `Normaliser()` est introduite pour réccupérer la solution réalisable du problème original, correspondant à la solution pénalisée.

Ce processus est réitéré pour un nombre d'itérations fixé au préalable. On peut également distinguer deux cas :

1. la solution courante est améliorée,
2. le procédé est incapable de produire une solution meilleure.

Concernant le premier cas (1), si l'étape est fixée à `phase normale`, alors la stratégie de pénalisation "`Pénaliser()`" est appliquée pour opérer des modifications sur la configuration de la solution en cours au niveau des valeurs profits de la fonction objectif. Sinon, nous sommes dans le cas (2), alors deux sous-cas se présentent :

- (i) la procédure "`Normaliser()`" est appliquée pour retransformer la solution pénalisée en une solution normale,
- (ii) la procédure "`Pénaliser()`" est appliquée à la dernière solution en cours.

```
Entrée : Une solution réalisable S et son évaluation O(S).
Sortie : Une solution réalisable S* et son évaluation O(S*).
Initalisation :
 Poser S* ⟵ S := CP() et V(ρ) ⟵ O(S*);
Etape générale :
 Tant que non (Condition_d'Arrêt()) faire
   S ⟵ CCP(S); /* Utiliser RechercheSwapLocal pour améliorer la solution initiale */
   Si V(ρ) ≤ O(S) alors
      Si (phase=Phase_Normale) alors /* Etape sans facteurs de pénalisation */
         Poser S* ⟵ S et V(ρ) ⟵ O(S*);
      Sinon
         Poser S= Normaliser(S, ρ);
            /* Remettre la solution dans sa configuration ordinaire */
         Poser S* ⟵ S et V(ρ) ⟵ O(S*);
      FinSi
   Sinon
      Si (phase=Phase_Normale) alors
         S ⟵ Pénaliser(V(ρ), Δ, π); /* Appliquer la pénalisation à la solution courante */
      Sinon /* Si l'étape courante est à Phase_Pénalisation */
         Poser S ⟵ Normaliser(S, ρ),  V(ρ) ⟵ O(S) et S ⟵ Pénaliser(V(ρ), Δ, π);
      FinSi
   FinSi
 FinTantQue

 Retourner S* et son évaluation O(S*);
```

FIG. 4.5 – Un algorithme avec pénalisation et normalisation : DerAlg

Notons que l'objectif de la dernière étape est de changer la trajectoire du processus de recherche pour explorer de nouvelles régions non visitées de l'espace des solutions réalisables.

Par ailleurs, l'algorithme DerAlg utilise d'autres paramètres intrinsèques. Le premier

paramètre est appelé *paramètre d'exploration* (noté Δ). Ce paramètre sert à fixer le nombre d'objets dont il faut pénaliser les valeurs profits v_{ij}, $i = 1, \ldots, n$, $j = 1, \ldots, r_i$. Le deuxième paramètre, appelé *paramètre diamètre* (noté D), est introduit pour contrôler et délimiter la région de l'espace à explorer pour améliorer la solution en cours. Pour cette raison, le dernier espace de recherche posséde quelques configurations différentes ayant la même valeur objective. Les derniers paramètres de l'approche sont les *paramètres de pénalisation* (notés π) appliqués à la fonction objectif. Ce faisant, chaque configuration représente une solution réalisable (valeur) pour le problème original.

Les principales étapes de l'algorithme Deralg sont données dans la figure 4.5. La complexité de l'algorithme DerAlg est donnée comme suit.
Initialement (`Initialisation`), DerAlg démarre en appliquant la procédure CP pour construire une solution initiale. Nous avons vu dans le paragraphe 4.2 que CP a une complexité au pire de $O(max\{\theta m, n\})$. Ensuite (`Etape générale`), on applique CCP pour améliorer la solution courante obtenue par CP. Sa complexité (voir la section 4.3) est en $O(\theta max\{m, n\})$. La procédure `Normaliser()` nécessite au pire cas θ opérations pour remettre une solution pénalisée dans sa configuration ordinaire et donc elle est en $O(\theta)$. La procédure `Pénaliser()` est aussi en $O(\theta)$, puisqu'elle utilise au pire θ opérations pour pénaliser une solution courante. Donc, l'algorithme DerAlg admet une complexité au pire cas égale à $MaxIter \times ((\theta m+n)+(\theta(m+n)\times\theta)$. D'où sa complexité est de l'ordre de $O(\theta^2 max\{n,\ m\})$.

4.7 Partie expérimentale

Dans cette section, nous allons exposer les résultas numériques obtenus par les approches. L'objectif de ces expériences est double : (i) d'une part, évaluer les performances des deux procédures constructive et complémentaire CP et CCP et, (ii) déterminer un meilleur compromis entre le temps d'exécution et le choix des paramètres intrinsèques utilisés par l'algorithme DerAlg : le nombre maximum d'itérations nécessaire à effectuer, le paramètre d'exploration Δ, le paramètre diamètre D et les paramètres de pénalité π.
Cette section est organisée comme suit. Tout d'abord, nous évaluons les performances des deux procédures CP et CCP. Ces procédures sont testées sur un ensembles d'ins-

tances de la littérature de différentes tailles et densités. Nous comparons les résultats obtenus par rapport à la solution optimale (ou la meilleure solution réalisable connue). Ces résultats sont ensuite comparés à ceux existant dans la littérature. Nous présentons ensuite les résultats obtenus par l'algorithme DerAlg et nous montrons l'importance des parmètres et de leur choix dans la résolution. Nous indiquons aussi le degré d'amélioration obtenu par l'algorithme DerAlg par rapport à d'autres approches de la littérature. Les méthodes CP, CCP et DerAlg ont été codés en `C++`, et exécutées sur une machine SUN `Ultra-Sparc10` (250 Mhz et 128Mo de RAM).

#Instance	n	r_i, $i = 1, \ldots, n$	m	N
I01	5	5	5	25
I02	5	10	5	50
I03	15	10	10	150
I04	20	10	10	200
I05	25	10	10	250
I06	30	10	10	300
I07	100	10	10	1000
I08	150	10	10	1500
I09	200	10	10	2000
I10	250	10	10	2500
I11	300	10	10	3000
I12	350	10	10	3500
I13	400	10	10	4000

TAB. 4.1 – Détails des instances

4.7.1 Performances des procédures CP et CCP

Pour évaluer les performances de CP et CCP, nous utilisons les instances proposées par Khan *et al.* [27]. Ces données décrivent une variété d'instances allant des petites tailles aux tailles plus importantes (nombre de variables variant entre $N = 25$ et $N = 4000$). La solution optimale de certaines de ces instances, notées `I1,...,I6` dans le Tableau 4.1, est connue. Pour les autres instances, notées `I7,...,I13` dans le Tableau 4.1, nous reportons la meilleure solution approchée publiée par Khan *et al.* [27]. Pour chaque instance, nous reportons le nombre n de classes, le nombre r_i d'éléments dans

chaque classe i, $i = 1, \ldots, n$, le nombre de contraintes m pour chaque instance et finalement le nombre total N d'éléments pour chaque instance, où $N = \sum_{i=1}^{n} r_i$.
Les résultats obtenus par Moser *et al.* [40] et Khan *et al.* [27] sont reportés dans le Tableau 4.2. Les résultats de Moser *et al.* sont représentés par les colonnes 5 et 6. Les résultats de l'algorithme de Khan *et al.* sont représentés par les colonnes 3 et 4. La colonne 2 contient la solution optimale (ou la meilleure solution approchée) du problème. Les colonnes 4 et 6 affichent le pourcentage de déviation de la valeur de la solution de l'optimum (ou de la meilleure valeur de la solution approchée). Nous notons ce pourcentage de déviation par Dev qui est calculé de la manière suivante :

$$Dev = \left(1 - \frac{A(I)}{Opt(I)(ou\ Best)}\right) \times 100,$$

où $A(I)$ et $Opt(I)$ (resp. $Best$) représentent la solution approchée (solutions des colonnes 3 et 5) et la solution optimale (resp. la meilleure solution approchée) de l'instance `I`.

#Instance	$Opt/Best$	$KLMA_{sol}$	Dev	$Moser_{sol}$	Dev
I01	173.00	167.00	3.47	151.00	12.72
I02	364.00	354.00	2.75	291.00	20.05
I03	1602.00	1533.00	4.31	1464.00	8.61
I04	3597.00	3437.00	4.45	3375.00	6.17
I05	3949.59	3899.10	1.28	3905.70	1.11
I06	4799.30	4799.30	0.00	4115.20	14.25
I07	23983.00*	23912.00	1.02	23556.00	2.50
I08	36007.00*	35979.00	0.11	35373.00	1.79
I09	48048.00*	47901.00	0.31	47205.00	1.75
I10	60176.00*	59811.00	0.68	58648.00	2.61
I11	72003.00*	71760.00	0.45	70532.00	2.16
I12	84160.00*	84141.00	0.03	82377.00	2.13
I13	96103.00*	96003.00	0.10	94166.00	2.02
Moyenne			1.46		5.99

TAB. 4.2 – Performance des approches de Khan *et al.*'s [27] et Moser *et al.*'s [40] sur les instances de la littérature. Le symbole $*$ signifie que la solution optimale n'est pas connue.

Dans le Tableau 4.3, les résultats obtenus par CP sont représentés par les colonnes 3, 4 et 5. La colonne 3 contient la valeur de la solution (notée CP_{sol}). La colonne 4 affiche le pourcentage de déviation (noté Dev) entre la valeur de la solution obtenue par CP et

la valeur de la solution optimale (ou la meilleure solution approchée), notée $Opt/Best$. La colonne 5 représente le temps d'exécution (noté T et mesuré en secondes) consommé par CP. Les colonnes 6 jusqu'à 8, quant à elles, reportent les résultats obtenus par la méthode CCP. La colonne 6 représente la solution obtenue par CCP (notée CCP_{sol}), la colonne 7 correspond à la déviation de la solution obtenue par rapport à l'optimum (ou de la meilleure solution), enfin la colonne 8 représente le temps d'exécution nécessaire à CCP pour s'arrêter avec une solution.

Cette section, peut être considérée comme une partie préliminaire dans laquelle nous comparons les résultats obtenus par CP à ceux obtenus par CCP.
Avant de comparer les résultats des deux algorithmes, nous analysons le comportement des deux approches de Moser *et al.* [40] et de Khan *et al.* [27]. A partir du Tableau 4.2, nous remarquons que la méthode de Khan *et al.* [27] (notée $KLMA_{sol}$) obtient des résultats meilleurs que la méthode proposée par Moser *et al.* [40] (dont les solutions sont notées par $Moser_{sol}$). Les résultats de Moser *et al.* sont tirés de Khan *et al.* [27]. Dans ce cas, $KLMA$ produit un pourcentage de déviation variant dans l'intervalle [0, 4.45%], et une moyenne de 1.46%.

#Instance	$Opt/Best$	CP_{sol}	Dev	T	CCP_{sol}	Dev	T
I01	173.00	161	6.94	<0.01	161.00	6.94	< 0.01
I02	364.00	284.00	21.98	<0.01	341.00	6.32	< 0.01
I03	1602.00	1414.00	11.74	<0.01	1511.00	5.68	< 0.01
I04	3597.00	3135.00	12.84	<0.01	3397.00	5.56	< 0.01
I05	3949.59	3065.40	22.40	<0.01	3591.59	9.06	0.03
I06	4799.30	3749.89	21.87	0.01	4567.90	4.82	0.02
I07	23983.00*	19667.00	18.59	0.02	23753.00	1.68	0.16
I08	36007.00*	28461.00	20.98	0.05	35485.00	1.48	0.40
I09	48048.00*	38389.00	20.10	0.06	47685.00	0.76	0.65
I10	60176.00*	48361.00	19.69	0.05	59492.00	1.21	1.11
I11	72003.00*	58008.00	19.53	0.08	71378.00	0.98	1.35
I12	84160.00*	68027.00	19.18	0.09	83293.00	1.04	1.70
I13	96103.00*	78309.00	18.52	0.09	95141.00	1.00	2.15
Moyenne			18.03	0.03		3.58	0.58

TAB. 4.3 – Performance des approches CP et CCP. Le symbole $*$ signifie que la solution optimale n'est pas connue. Le symbole $<$ signifie que le temps d'exécution T est négligeable.

Maintenant, pour analyser le comportement de CP et de CCP, nous avons résumé les résultats de ces deux approches dans le Tableau 4.3. On peut remarquer qu'avec

CCP, on obtient des résultats nettement meilleurs que ceux obtenus par CP. Par ailleurs, CCP nécessite un temps d'exécution plus imporant. Néanmoins, CCP produit de meilleures solutions (une déviation moyenne égale à 3.06%) en consommant moins que 0.58 *secondes* en moyenne. Occasionnellement, elle produit des résultats d'une qualité médiocre (au pire cas d'un pourcentage de 8.93%). Cependant, CCP demeure une procédure de démarrage utile pour des procédures plus élaborées que nous verrons par la suite.

4.7.2 Performance de l'algorithme DerAlg

Généralement, si on utilise des algorithmes approchés pour résoudre des problèmes d'optimisation, il est bien connu que différents paramètres et leur choix prennent part dans la qualité de la solution obtenue. Pour ce qui concerne notre approche, DerAlg implique quatre décisions : (i) la manière de choisir le paramètre d'exploration Δ, (ii) le nombre d'itérations $MaxIter$ à fixer, (iii) la manière de contrôler l'espace de recherche à délimiter par le paramètre diamètre D et finalement, (iv) les valeurs à attribuer aux paramètres de pénalisation π (nous avons choisi d'affecter la même pénalisation à toutes les valeurs profits).

Dans la suite, différents ajustements sont opérés pour tenter d'atteindre de meilleures solutions. Cependant, le meilleur ajustement retenu, peut entraîner un temps d'exécution plus important. Nous avons donc fait un choix de compromis entre la qualité de la solution finale et le temps d'exécution nécessaire.

Afin de trouver une meilleure valeur à attribuer au paramètre exploration Δ, nous avons exploré trois stratégies :

- affecter à Δ une valeur large, en fixant ce paramètre dans l'intervalle discret $[6, \ldots, 10]$;
- affecter à Δ une valeur intermédiaire, en fixant ce paramètre à 5 ;
- affecter à Δ une valeur faible, en fixant ce paramètre dans l'intervalle discret $[1, \ldots, 4]$.

Les résultats obtenus montrent que la variation de Δ dans l'intervalle $[1, \ldots, 4]$ produit des solutions de meilleures qualité. Pour l'intervalle complémentaire, $[6, \ldots, 10]$, l'algorithme n'était pas capable de produire de meilleures solutions sinon d'aussi bonne qualité. En revanche, pour cet intervalle, la consommation en terme de temps d'exécution

est plus importante. Enfin, les meilleurs résultats ont été obtenus pour le deuxième cas où Δ a été fixé à la valeur 5.

Afin de trouver un meilleur compromis entre la qualité de la solution et le temps d'exécution, nous avons introduit une variation sur le nombre maximum d'itérations $MaxIter$. Dans ce cas, nous avons testé $MaxIter$ avec des valeurs prises dans l'intervalle discret [2, 5, 8, 10]. Les résultats obtenus montrent que les meilleures solutions ne sont pas nécessairement obtenues pour de grandes valeurs de $MaxIter$ et de ce fait, le temps d'exécution de DerAlg croît aussi.

MaxIter	2	5	8	10
Av.Dev	1.81	0.92	0.68	0.61
Av.T	1.90	1.90	4.10	6.50

TAB. 4.4 – Le comportement de DerAlg par rapport à la variation du paramètre $MaxIter$.

Le Tableau 4.4 représente la qualité des solutions obtenues en fonction des valeurs données suivantes : $\Delta = 5$, $D = 5$ et $\pi = 0.70$ (dans la suite, nous discuterons le choix des valeurs associées à D et à π). Utilisant ces valeurs, telles que c'est montré dans le Tableau 4.4, nous remarquons que la qualité des solutions obtenues (notée par Av.Dev. de la ligne 2) varie entre 0.61% et 1.81%. La meilleure déviation moyenne est obtenue pour $MaxIter = 10$ en un temps d'exécution moyen le plus important (notée Av.T. de la ligne 3).

En fixant les valeurs des paramètres Δ et $MaxIter$, on cherche maintenant la meilleure stratégie pour fixer la valeur du paramètre diamètre D. Rappelons que D est appliqué pour délimiter l'espace de recherche. En effet, D permet d'effectuer une certaine diversification de la solution lorsque plusieurs meilleures solutions (par exemple, ces solutions ont la même valeur objective mais possèdent des configurations différentes) sont atteintes par l'algorithme DerAlg. Le Tableau 4.5 reporte la qualité des solutions obtenues lorsque D varie dans l'intervalle discret [3, 5, 7, 10].

On peut remarquer que la déviation moyenne varie entre 0.61% et 0.70%, et le meilleur résultat est obtenu pour $D = 5$. Le même Tableau 4.5, montre que si la valeur de D est assez petite ou assez grande, alors la diversification utilisée n'est pas assez suffisante ou trop importante. Nous pensons que pour de faibles valeurs de D, l'espace

D	3	5	7	10
Av.Dev	1.14	0.61	0.63	0.70
Av.T	2.50	6.50	7.40	9.50

Tab. 4.5 – Effets du paramètre D sur l'algorithme DerAlg.

de recherche généré n'est pas assez riche ou peu suffisant pour explorer de nouvelles solutions. En revanche, pour des valeurs assez grandes de D, l'espace de recherche est très vaste de telle façon que l'algorithme est incapable de localiser la meilleure direction de recherche pour améliorer les solutions déjà visitées. A partir du Tableau 4.5, on peut conclure qu'une valeur intermédiaire de D assure une qualité supérieure de la solution obtenue.

π	0.50	0.70	0.80	0.90
Av.Dev	0.70	0.61	0.64	0.70
Av.T	6.40	6.50	6.90	9.50

Tab. 4.6 – Le comportement de DerAlg quand on fait varier π.

Finalement, nous analysons le comportement de l'algorithme DerAlg lorsqu'on fait varier la valeur π du paramètre de pénalisation. Le Tableau 4.6 résume les résultats obtenus pour différentes valeurs affectées à π. Dans ce tableau, on peut remarquer que DerAlg réalise de meilleurs résultats avec la valeur $\pi = 0.70$. Ce choix entraîne une déviation moyenne égale à 0.61%. Notons que pour d'autres valeurs données de π, l'algorithme dégrade la qualité de la solution. Ajouté à celà, DerAlg est très rapide lorsque $\pi = 0.70$. L'algorithme nécessite un temps d'exécution moyen égal à 6.5 *secondes*.
Nous pouvons par ailleurs conclure qu'il est nécessaire d'utiliser de faibles valeurs ou de grandes valeurs de π (proches de la valeur 0.5 ou de la valeur 1) pour espérer obtenir des solutions de meilleure qualité.
Dans ce qui suit, nous donnons les solutions produites par DerAlg et nous comparons sa performance à celle de Khan *et al.*, notée par KLMA (voir Tableau 4.2). Nous considérons spécifiquement la version de l'algorithme pour laquelle les paramètres suivants ont été choisis : $\Delta = 5$, $D = 5$, $\pi = 0.70$ et $MaxIter = 10$.

La performance de DerAlg est déterminée sur l'ensemble des instances de la littérature (voir Tableau 4.1). Les résultats de l'algorithme sont reportés dans le Tableau 4.7.

#Instance	$KLMA_{sol}$	$DerAlg_{sol}$	Dev	T
I01	167.00	173°	-3.59	0.04
I02	354.00	356.00$^\triangleright$	-2.82	0.04
I03	1533.00	1553.00$^\triangleright$	0.00	0.08
I04	3437.00	3502.00$^\triangleright$	-1,89	0.09
I05	3899.10	3943.22$^\triangleright$	-1.13	0.15
I06	4799.30	4799.30	0.00	0.21
I07	23912.00*	23983.00°	-0.30	1.50
I08	35979.00*	36007.00°	-0,08	2.17
I09	47901.00*	48048.00°	-0.31	5.50
I10	59811.00*	60176.00°	-0.61	7.47
I11	71760.00*	72003.00°	-0.34	13.35
I12	84141.00*	84160.00°	-0.02	22.41
I13	96003.00*	96103.00°	-0.10	31.64
Moyenne			-0.86	6.50

TAB. 4.7 – Performance de DerAlg testé sur les instances du Tableau 4.1 et comparé aux résultats de KLMA. Le symbole $\circ$ signifie que la solution optimale (ou la meilleure) est atteinte et le symbole $\triangleright$ signifie que DerAlg améliore la solution produite par KLMA.

Pour chaque instance, nous reportons la valeur de la solution obtenue (notée $DerAlg_{sol}$), la déviation (noté Dev) entre la solution obtenue par DerAlg et celle donnée par KLMA. Dans ce cas, la déviation négative $-\xi$ signifie que l'algorithme améliore de $\xi\%$ la solution existante. Nous reportons également le temps d'exécution T (mesuré en secondes) et la déviation moyenne (resp. le temps d'exécution moyen) que DerAlg consomme pour produire la solution finale (la dernière ligne du Tableau 4.7). Dans cette partie, nous nous intéressons principalement à la qualité des solutions obtenues.

Le Tableau 4.7 montre que DerAlg produit de meilleures solutions que KLMA. En moyenne, il réalise un pourcentage d'amélioration (comparées aux solutions produites par KLMA) de l'ordre de 0.86%. Ce pourcentage varie dans l'intervalle $[0, 3.59]$ sur l'ensemble des instances traîtées.

Par ailleurs, on peut remarquer que les solutions sont obtenues dans moins d'une minute ($T \leq 60$ *secondes*) et ceci, spécialement pour les instances de grande taille. Notons de plus que, pour les petites instances `I1`,...,`I6`, DerAlg améliore d'une manière significative les solutions produites par CCP et produit de meilleurs résultats comparés à ceux donnés par KLMA. Pour les instances de grande taille `I7`,..., `I13` et pour lesquelles nous ne disposons pas de solutions optimales, DerAlg produit de meilleures solutions comparées à celles produites par KLMA.

La figure 4.6 illustre le pourcentage d'amélioration entre la solution produite par CCP et DerAlg quand on utilise les paramètres suivant : $\Delta = 5,\ D = 5,\ \pi = 0.70$ et $MaxIter = 10$. La figure 4.7 montre le comportementr de DerAlg comparé à celui de KLMA. On peut remarquer que DerAlg se comporte mieux que KLMA dans le sens qu'il produit des solutions de meilleure qualité.

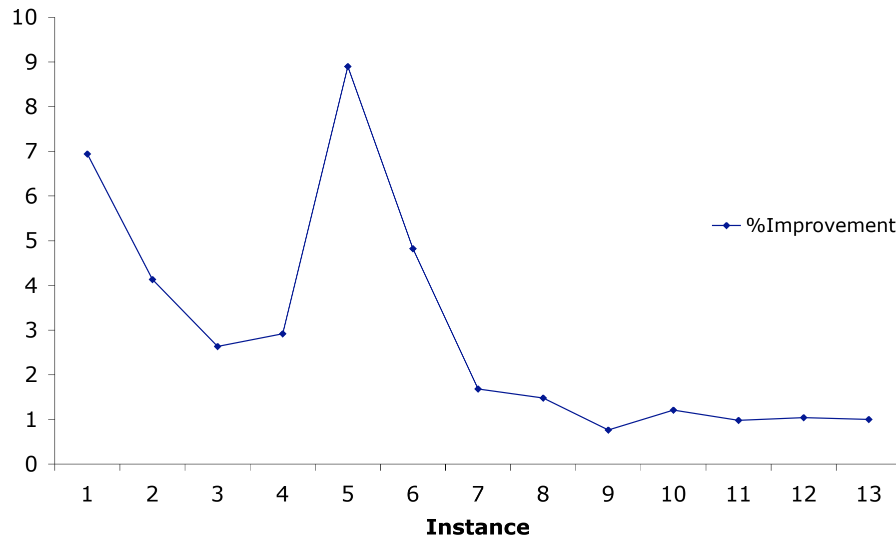

Fig. 4.6 – Variation du pourcentage d'amélioration de DerAlg versus CCP.

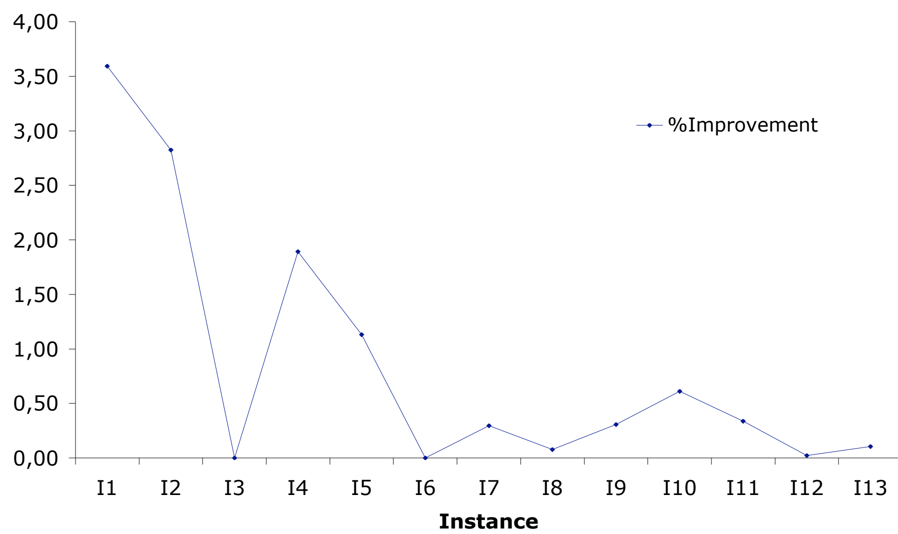

FIG. 4.7 – Variation du pourcentage de déviation de KLMA et DerAlg.

Partie B : la recherche locale réactive

4.8 Une recherche locale réactive

Pour un problème d'optimisation combinatoire, les algorithmes de recherche locale sont généralement décrits sous forme de plusieurs composantes. Parmi ces composantes, on peut distinguer (en général) : (*i*) le *problème combinatoire* à résoudre, (*ii*) la représentation d'une *fonction coût* (ou objective) associée à une instance du problème d'une part, et un *domaine de voisinage* qui définit les principales transitions dans l'espace de recherche d'autre part, et (*iii*) la stratégie de contrôle pour les *mouvements locaux* à entreprendre. Plusieurs stratégies ont été proposées pour permettre au problème de dépasser les optimas locaux susceptibles de bloquer la recherche de meilleures solutions. Souvent, les mouvements ne permettant pas l'amélioration locale sont, soit basés sur une décision probabiliste (*bruit*), soit sur l'historique de la recherche.

4.8.1 Un algorithme de recherche locale réactive

Des heuristiques dédiées à la résolution des problèmes d'optimisation combinatoire peuvent utiliser la structure générale de la recherche locale réactive. La recherche locale est souvent complétée par un processus (dit *réactif*), s'appuyant sur l'historique de la recherche pour augmenter son efficacité.

Dans notre étude, nous proposons une approche qui s'inspire de la recherche locale réactive qui consiste à utiliser : (*i*) une rétroaction simple pour agir sur la valeur du paramètre d'interdiction en recherche tabou et, (*ii*) deux stratégies différentes appliquant ou bien une procédure notée *débloquer* ou bien une *liste de stockage local*. La stratégie du déblocage consiste à débloquer le processus de recherche si la solution semble cycler, et la liste de stockage local est utilisée dans le but de (*a*) localiser les solutions indésirables d'une part, et (*b*) d'interdire toute configuration ayant la même valeur solution d'autre part.

4.8.2 Représentation d'une solution

Nous avons donné dans la section 4.3 une représentation standard d'une solution représentant le MMKP. Nous rappelons qu'une solution peut être donnée comme suit :

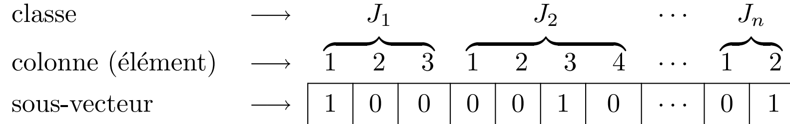

FIG. 4.8 – Représentation d'une solution du MMKP.

Rappelons aussi que pour chaque classe J_i, $i = 1, \ldots, n$, on séléctionne un et un seul élément j, c'est-à-dire, $x_{ij} = 1$ si l'élément j de la classe J_i est séléctionné, $x_{ij} = 0$ sinon.
Par ailleurs, nous utiliserons la notion des états ER et EN de la section 4.3 et les deux procédures : constructive de la section 4.3.1, et complémentaire de la section 4.3.3. Notons que les classes J_i, $i = 1, \ldots, n$ forment une partition (réunion disjointe) de l'ensemble $\mathcal{N}$ de tous les objets.

Comme nous l'avons décrit dans la section 4.3, nous distingueons deux différents états : *état réalisable* (ER) et *état non réalisable* (EN) :

$$(ER): \ \forall k \in \{1, \ldots, m\}, \ \sum_{i=1}^{n} \sum_{j=1}^{r_i} w_{ij}^k x_{ij} \leq C^k;$$

$$(EN): \ \exists k \in \{1, \ldots, m\}, \ \sum_{i=1}^{n} \sum_{j=1}^{r_i} w_{ij}^k x_{ij} > C^k.$$

L'objectif est de construire des (ER) améliorés (ou de transformer un état EN en ER) en appliquant une stratégie de recherche et d'*échange local.*

4.8.3 Principe de la recherche locale adaptée

Cette section, décrit la première approche de recherche réactive pour le MMKP. Le but de la méthode est d'améliorer la solution produite par la procédure CCP (voir section 4.3.3). L'algorithme, (noté RLS), utilise un paramètre prévoyant les cycles pour indiquer si la solution en cours cycle et de réagir sur cette dernière si nécessaire. La

méthode 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.

4.8.4 Description de l'algorithme

La figure 4.9 décrit les principales étapes de l'algorithme. L'algorithme démarre en appliquant la procédure CP pour construire une solution de départ. Chaque solution est représentée par sa configuration stockée dans un vecteur ρ. La meilleure solution (resp. évaluation) S^* (resp. $O(S^*)$) est initialement égale à S (resp. $O(S)$) atteinte par CCP.

```
Entrée : Une solution réalisable S et son évaluationO(S).
Output : Une solution réalisable améliorée S* et son évaluation O(S*).
--------------------------------------------------------------------
Initalisation.
 Poser S* := S :=argmax {CP_sol, CCP_sol}, et poser V(ρ) := O(S*);
Etape générale.
 Tant que non (Condition_d'Arrêt()) faire
   Poser p := 0; /* p est le nombre de fois que l'on peut dégrader une solution en cours*/
     Pour i dans J faire :
       k̄_i := RechercheSwapLocal(k, J_i);
            /* k (k̄_i) définit l'ancien (nouvel) élément de la classe J_i */
       Si O(S_{k̄_i}) > O(S) alors S := S_{k̄_i};
     FinFaire
     Si V(ρ) < O(S) alors S* := S, V(ρ) := O(S*) et p := 0;
     Sinon
       Si (p < Const) alors S := Dégrader(S) et poser p := p + 1;
       Sinon
            S :=Débloquer(S);
            Si O(S) > V(ρ) alors p := 0;
            Sinon sortir avec la meilleure solution en cours;
 FinTant Que
 Sortir avec la meilleure solution S* (de valeur O(S*)).
```

FIG. 4.9 – L'algorithme RLS pour le MMKP.

Ensuite, dans la boucle générale, on applique la procédure notée dans l'algorithme par `RechercheSwapLocal()` pour obtenir une première solution améliorée. La meilleure solution est mise à jour si la solution obtenue réalise la plus grande valeur comparée à la valeur initiale. Ce procédé est réitéré pour un nombre maximum d'itérations.

Si la procédure `RechercheSwapLocal()` ne parvient pas à améliorer la solution, alors la stratégie de dégradation d'une solution, `Dégrader()`, est introduite pour agir sur la solution non améliorée. L'objectif de cette opération et de changer la trajectoire de recherche vers une autre région. Ensuite, on réactive le processus de recherche en appliquant à nouveau la procédure `RechercheSwapLocal()`. Cette stratégie est répétée pour un nombre fixé d'itérations.
Comme c'est décrit dans l'une des étapes de l'algorithme, si la solution obtenue semble cycler et n'est pas "suffisamment" améliorée, on évite ce phénomène en utilisant la stratégie de déblocage d'une solution, `Débloquer()`. La procédure `RechercheSwapLocal()` est appelée à nouveau pour améliorer la solution débloquée. Ce processus est contrôlé par un nombre maximum d'itérations.

De point de vue complexité, RLS est en $O(\theta nm^2)$, où $\theta = max\{r_1, \ldots, r_n\}$.
En effet, RLS démarre par un appel de CP de complexité $O(max\{m\theta, n\})$ (voir section 4.3.1). Ensuite, RLS fait appel à la procédure `RechercheSwapLocal()` de compléxité $O(\theta nm)$ et les deux procédures `Dégrader()` et `Débloquer()` qui sont en $O(m)$ chacune. Finalement, la complexité totale est de l'ordre de $MaxIter \times ((m\theta + n) + 2(nm\theta)) \times m$. Par conséquent, la complexité au pire de RLS est en $O(\theta nm^2)$.

4.9 Un algorithme modifié de recherche locale et réactive

Dans cette section, nous proposons une modification de l'algorithme RLS de la section 4.8.3. Nous introduisons une liste *mémoire* afin de tenter de prévenir le cyclage des solutions. Notons, qu'en appliquant l'algorithme RLS au problème MMKP, on peut obtenir différentes configurations de même évaluation. Dans ce cas, après qu'un mouvement ne soit effectué, le processus vérifie si la configuration courante a déjà été obtenue durant la recherche et réagit en conséquence. Ce mécanisme réactif n'est sans doûte pas suffisant pour garantir à la trajectoire de recherche de n'être pas bloquée dans une

région limitée de l'espace de recherche. La robustesse de l'approche requiert alors un autre mécanisme d'ésquive d'un tel phénomène. Nous l'appelons *remplissage-mémoire*. Cette stratégie est appliquée lorsque différentes configurations obtenues durant la recherche améliorante, réalisent la même valeur de la fonction objective.

La figure 4.10 décrit les principales étapes de l'algorithme modifié (noté MRLS).

Entrée : Une solution réalisable S de valeur $O(S)$.
Sortie : Une solution réalisable améliorée S^* de valeur $O(S^*)$.

Initalisation :
Poser $S^* := S :=\text{argmax}\left\{\text{CP}_{sol}, \text{CCP}_{sol}\right\}$, et poser $V(\rho) := O(S^*)$;
Poser $Liste = \emptyset$;
Etape générale :
Tant que non (**Condition_d'Arrêt()**) **faire**
  mettre $p := 0$ /* p : nombre de fois une solution courante peut être dégradée */
  Répéter
    Pour $i \in \mathcal{J}$ **faire**
      /* k ($\bar{k}_i$) représente l'ancien (nouvel) élément de la classe J_i */
      $\bar{k}_i :=$ **RechercheSwapLocal**(k, J_i) et $S_{\bar{k}_i} \notin Liste$;
      Si $O(S_{\bar{k}_i}) > O(S)$ **alors** $S := S_{\bar{k}_i}$;
    FinFaire ;
    Si $V(\rho) < O(S)$ **alors** $S^* := S,\ V(\rho) := O(S^*)$ et $p := 0$;
    Sinon
      $p := p + 1$;
      Si $(p < Const)$ **alors**
        Insérer$(S, Liste)$;
        $S :=$ **Dégrader**(S) et $S \notin Liste$;
      FinSi ;
  Jusqu'à $(p = Const)$;
FinTant Que ;
Sortir avec la meilleure solution S^* de valeur $O(S^*)$.

FIG. 4.10 – Modification de l'algorithme RLS en ajoutant une liste mémoire : MRLS.

Nous proposons donc de modifier la première version de la recherche locale, RLS, en choisissant de remplacer la stratégie de déblocage d'une solution par la liste mémoire. Ce changement est effectué sur les solutions dont les valeurs ne sont pas suffisemment améliorées. Au départ, et après avoir effectué une recherche d'interchange local, la meilleure solution en cours est mémorisée dans la liste appelée *mémoire* (notée *Liste*). Le but alors, est de localiser des solutions équivalentes (configurations différentes avec la même évaluation) et d'interdire tout retour à ces solutions. Ensuite, la stratégie de dégradation d'une solution est appliquée pour un certain nombre d'itérations. Si la

solution obtenue n'est pas suffisamment améliorée, alors sa configuration est stockée dans la liste mémoire, ainsi on permet à la recherche de changer de direction.

En phase d'initialisation, l'algorithme MRLS démarre en faisant appel à la procédure CP afin d'obtenir une solution initiale. La configuration de chaque solution est représentée par un vecteur noté ρ. La meilleure solution (resp. évaluation) S^* $(resp.O(S^*))$ est initialement fixée à la solution initiale (resp. évaluation initiale) S $(resp.O(S))$ obtenue par CP et améliorée par CCP. Egalement, la liste mémoire *Liste* est initialement vide.

Dans la phase de l'étape générale, l'algorithme effectue des interchanges locaux afin d'améliorer la solution courante qui n'appartient pas à la liste mémoire. Ensuite, la solution obtenue est comparée à la valeur stockée. Si la valeur de celle-ci améliore l'ancienne valeur, alors elle devient la meilleure solution obtenue. Ce processus est réitéré pour un nombre fixé d'itérations. Dans ce cas, la solution courante non améliorée est introduite dans la liste pour éviter au processus de recherche de solutions de cycler.

On applique, ensuite, la stratégie de dégradation de la solution pour permettre à celle-ci de réagir par diversification. Dans ce cas, la procédure de recherche d'interchange local est réappliquée pour améliorer la solution après rétroaction.

De point de vue complexité, on remarque que MRLS admet une même complexité que RLS, dans le sens qu'il utilise les mêmes procédures excépté le fait qu'à la place de la procédure de déblocage de la solution, il utilise une liste mémoire. Donc, sa complexité au pire est en $O(\theta^2(m+n))$ (voir section 4.5 pour plus de détails).

4.10 Partie expérimentale

Les deux versions de l'approche sont codées en `C++` et exécutées sur une station SUN `Ultra-Sparc10` (250 Mhz et avec 128Mo de mémoire RAM). Ces deux versions ont été testées sur le même ensemble d'instances du paragraphe 4.10 proposé par Khan *et al.* [27, 28].

4.10.1 Comportement des deux versions de l'approche

Généralement, quand on opte pour une stratégie de type heuristique pour résoudre un problème d'optimisation, il est important de bien choisir les paramètres intrinsèques à l'approche développée pour permettre d'obtenir de meilleures solutions. Concernant

notre approche, le même principe s'applique à RLS et à MRLS qui nécessitent quatre décisions.

- RLS utilise le paramètre p qui correspond au nombre de fois qu'une solution courante peut être dégradée et $MaxIter$ le nombre d'itérations autorisées pour l'interchange local.
- MRLS utilise en plus des deux paramètres cités ci-dessus, une liste mémoire dont la taille doit être fixée.

Un meilleur choix de ces paramètres, permettraient d'obtenir des solutions de qualité "supérieure", cependant, ces ajustements conduisent parfois à des temps d'exécution plus importants. L'ensemble des valeurs choisies pour nos tests représente un meilleur compromis entre la qualité de la solution obtenue et le temps nécessaire à l'obtention d'une telle solution pour une instance du problème MMKP. Ci-après nous présentons les différents réglages utilisés par les deux versions de l'algorithme.

Pour obtenir une meilleure valeur pour $MaxIter$, représentant le nombre maximum d'itérations utilisé par l'algorithme, nous avons introduit une variation de ce paramètre dans l'intervalle discret $\{5n,\ 15n,\ 20n\}$. Ces tests ont été effectués en fixant le paramètre p, qui autorise le nombre de fois qu'une solution courante peut être dégradée, à 5 (plus loin, nous discuterons le choix du paramètre p).
Lors des tests, nous avons constaté que lorsque $MaxIter$ prenait des valeurs grandes, alors (i) la qualité des solutions devenait meilleures et (ii) le temps d'exécution devenait très important. Le Tableau 4.8 montre la qualité des résultats obtenus lorsqu'on fait varier le nombre d'itérations. On peut remarquer que la moyenne du rapport d'approximation (AR : colonne 2) varie entre 0.9752 et 0.9974. La meilleure moyenne de AR est obtenue en fixant la valeur de $MaxIter$ à $15n$ avec un temps d'exécution moyen le plus important (Ligne 3, colonne 3) et en obtenant le plus grand nombre de solutions optimales/meilleures (Ligne 3, colonne 4). Dans ce qui suivra, on fixera $MaxIter$ à $15n$.

Maintenant, discutons du choix du paramètre p. Pour cela on analyse le comportement de RLS lorsqu' on fait varier la valeur de p. Le Tableau 4.9 reporte la qualité des solutions obtenues pour p choisis dans l'intervalle discret $\{5,\ 15,\ 20\}$. On peut constater que la moyenne du rapport d'approximation (colonne 2) varie entre 0.9812 et 0.9974, et que le meilleur résultat est obtenu pour $p = 5$. Le tableau montre aussi que si on opte pour une valeur de p aussi grande, alors la diversification utilisée devient

# Itérations	Moyenne AR	Moyenne T	# Solutions optimales/meilleures
5n	0.9752	2.42	7
10n	0.9886	6.14	7
15n	0.9974	8.04	8

Tab. 4.8 – Comportement de RLS selon la variation du nombre d'itérations $MaxIter$ et en fixant p à 5.

moins significative (dans le sens que l'approche est incapable de produire de meilleures solutions). Pour une valeur de $p = 5$, RLS consomme en moyenne un temps moins important (Ligne 3, colonne 4) et produit un nombre important de solutions optimales ou meilleures. Néanmoins, nous pensons que pour la plus grande valeur, l'algorithme explore une région plus importante de l'espace admissible et par conséquent, la recherche réactive est incapable de repérer une meilleure direction de recherche pour améliorer des solutions visitées. On peut donc conclure qu'une valeur intermédiaire pour p permet d'obtenir de meilleures solutions.

Varier p	Moyenne AR	Moyenne T	# Solutions optimales/meilleures
5	0.9974	8.04	8
15	0.9865	8.32	7
20	0.9812	9.04	7

Tab. 4.9 – Comportement de RLS selon la variation de p.

Nous discutons à présent, le choix de la taille de la mémoire, notée $Liste$, utilisée dans la seconde version de l'algorithme MRLS. Nous rappelons que la liste mémoire remplace la stratégie de déblocage d'une solution dans la version MRLS de l'approche. Ce remplacement est effectué sur les solutions qui ne sont pas suffisamment améliorées. Dans notre étude, nous avons considéré une variation dynamique de la longueur de la liste. En effet, si n représente le nombre de classes, alors la longueur de la liste est automatiquement choisie dans un intervalle discret. Le changement dans la liste est effectué après 50 itérations sans amélioration de la meilleure solution courante. Afin de trouver un meilleur intervalle de variation de la longueur de la liste, nous avons comparé différents intervalles. Le Tableau 4.10 reporte les résultats obtenus par MRLS pour les différents choix d'intervalles (sur l'ensemble des tests que nous avons effectués,

nous avons reporté trois types d'intervalles significatifs).

On peut remarquer que le rapport d'approximation moyen varie entre 0.9921 et 1, et les meilleurs résultats sont obtenus pour l'intervalle $[2n,\ 2n+10]$. Dans ce cas, MRLS atteint toutes les solutions optimales/meilleures et consomme un temps d'exécution moyen égal à 14.52 *secondes*. On remarque aussi que si la taille de la mémoire est grande ou petite, alors le stockage des solutions devient moins efficace. En effet, MRLS dégrade la qualité des résultats obtenus. On pense que pour un petit intervalle, la mémoire est incapable de détecter certains phénomènes liés au cyclage. En revanche, pour un intervalle de taille importante, la mémoire stocke un nombre important de valeurs correspondant aux configurations (ce qui empèche une meilleure exploration de l'espace). Ajouté à cela, le traitement devient "lourd", ceci est dû à la recherche d'une valeur dans la mémoire. On peut conclure qu'il n'est pas nécessaire d'utiliser une petite ou une grande taille de la mémoire pour obtenir de meilleures solutions.

Intervalle de la liste	Moyenne AR	Moyenne T	# Solutions optimales/meilleures
$[n, n+10]$	0.9959	8.11	7
$[2n, 2n+10]$	1	14.52	10
$[3n, 3n+10]$	0.9921	16.23	8

TAB. 4.10 – Comportement de MRLS selon la variation dynamique de la longueur de la liste mémoire.

4.10.2 Performance des approches RLS et MRLS

Dans ce paragraphe, nous comparons les performances des différentes approches DerAlg, RLS et MRLS. Par la suite, nous donnons une étude détaillée des résultats obtenus.

Dans le Tableau 4.11, on présente une étude comparative des approches DerAlg, RLS et MRLS. Le détail des résultats numériques des deux versions de l'approche (RLS et MRLS) est reporté dans le Tableau 4.12. Dans les deux tableaux, nous reportons la solution optimale (pour certaines instances) ou la meilleure solution connue (pour des instances dont on ne dispose pas de solutions optimales). On les notera $Opt/Meilleure$. On reporte aussi la valeur de la solution obtenue (qu'on note $DerAlg_{sol}$, RLS_{sol}, et

$MRLS_{sol}$) et le temps d'exécution consommé par RLS et MRLS (noté T et mesuré en *secondes*).

Le Tableau 4.11 reporte l'étude comparative de DerAlg, RLS et MRLS. On peut remarquer des Tableaux 4.11 et 4.12 que :

1. RLS améliore les résultats obtenus par l'approche proposée dans la section 4.7, noté DerAlg (voir Tableau 4.11, colonnes au dessous de RLS_{sol} marquées par le symbole $\circ$). En effet, on peut remarquer (voir Tableau 4.12, colonnes 3 et 4) que RLS améliore toutes les solutions en consommant un temps d'exécution évalué en moyenne à moins d'une minute (plus grand temps $T \leq 20$ *sec*).

Inst.	$Opt/Meilleure$	$DerAlg_{sol}$	RLS_{sol}	$MRLS_{sol}$
I01	173.00	173.00	173.00	173.00°
I02	364.00	356.00	364.00	364.00°
I03	1602.00	1553.00	1595.00°	1602.00°
I04	3597.00	3502.00	3564.00°	3569.00$^{\circ\diamond}$
I05	3949.59	3943.22	3945.27°	3945.27$^{\circ\diamond}$
I06	4799.30	4799.30	4799.30	4799.30
I07	24159.00*	23983.00	24121.00°	24159.00°
I08	36401.00*	36007.00	36110.00	36401.00°
I09	48367.00*	48048.00	48291.00$^{\circ\diamond}$	48367.00°
I10	60475.00*	60176.00	60291.00°	60475.00$^{\circ\diamond}$
I11	72558.00*	72003.00	72283.00$^{\circ\diamond}$	72558.00°
I12	84707.00*	84160.00	84446.00°	84707.00$^{\circ\diamond}$
I13	96834.00*	96103.00	96580.00°	96834.00$^{\circ\diamond}$

TAB. 4.11 – Sommaire des résultats obtenus par DerAlg, RLS et MRLS. Le symbole $*$ signifie que la solution optimale n'est pas connue.

Inst.	$Opt/Meil$	RLS_{sol}	T	$MRLS_{sol}$	T
I01	173.00	173.00	0.05	173.00°	0.56
I02	364.00	364.00	0.07	364.00°	0.79
I03	1602.00	1595.00°	0.20	1602.00°	1.99
I04	3597.00	3564.00°	0.22	3569.00°◇	2.28
I05	3949.59	3945.27°	0.19	3945.27°◇	1.93
I06	4799.30	4799.30	0.23	4799.30	2.36
I07	24159.00*	24121.00°	1.20	24159.00°	11.66
I08	36401.00*	36110.00	2.37	36401.00°	**14.93**
I09	48367.00*	48291.00°◇	4.33	48367.00°	20.03
I10	60475.00*	60291.00°	6.65	60475.00°◇	25.45
I11	72558.00*	72283.00°◇	9.52	72558.00°	30.27
I12	84707.00*	84446.00°	12.69	84707.00°◇	36.32
I13	96834.00*	96580.00°	16.73	96834.00°◇	**41.20**
Moyenne			4.18		14.59

TAB. 4.12 – Résultats numériques obtenus par RLS et MRLS. Le symbole $*$ signifie que la solution optimale n'est pas connue.

2. Dans certains cas, les résultats de MRLS sont améliorés (utilisant la mémoire notée *Liste* - voir Tableau 4.11, colonne 6 : marquée par le symbole $\diamond$). Les solutions obtenues sont de meilleure qualité en les comparant aux solutions obtenues par RLS. Globalement, la moyenne des résultats est meilleure que la qualité moyenne des résultats obtenus par RLS. Plus concrètement, si on applique l'approche MRLS, le nombre de solutions optimales/meilleures obtenu par RLS augmente de 9 à 12 solutions et le pourcentage des solutions améliorées augmente significativement de 69.24% à 92.31%. Dans ce cas, on peut constater que MRLS, obtenu par hybridation de l'approche tabou et d'autres stratégies, produit de meilleure solutions.

3. Notons que MRLS par comparaison à RLS nécessite parfois un temps d'exécution plus important. Néanmoins, il obtient des solutions de meilleure qualité même dans les cas les moins favorables. Ces solutions sont obtenues en un temps d'exécution moyen inférieur à 15 *secondes* et, en moins de 42 *secondes* pour les instances de grande taille.

4. Les graphiques de la Figure 4.11 montrent le comportement des deux méthodes

RLS et MRLS pour toutes les instances traitées. Pour chacune des deux versions, on peut remarquer que le temps d'exécution consommé croit avec la taille de l'instance du problème à résoudre. La figure montre aussi que l'accroissement de la courbe pour MRLS est plus importante que pour RLS. Comme illustré dans les graphiques de la Figure 4.12, l'accroissement du temps d'exécution est compensé par la qualité bien meilleure des résultats obtenus par MRLS (en particulier pour les instances de grande taille).

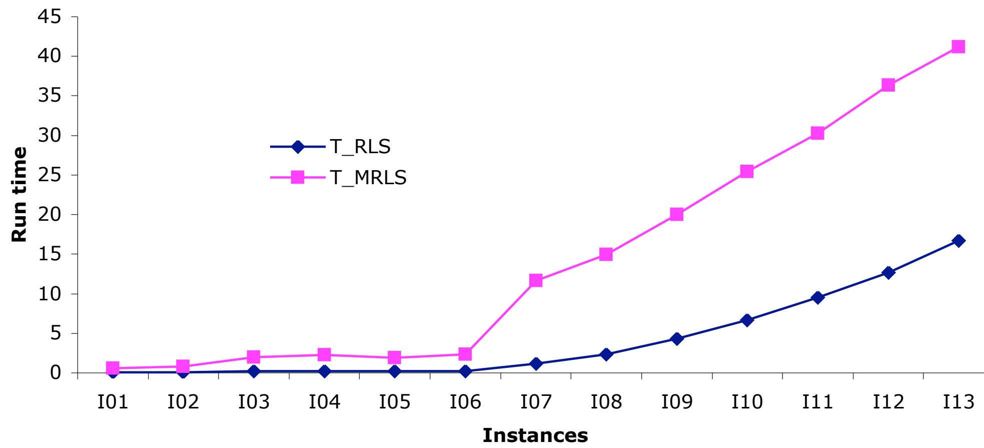

FIG. 4.11 – Comparaison des temps d'exécution consommés par les approches RLS et MRLS.

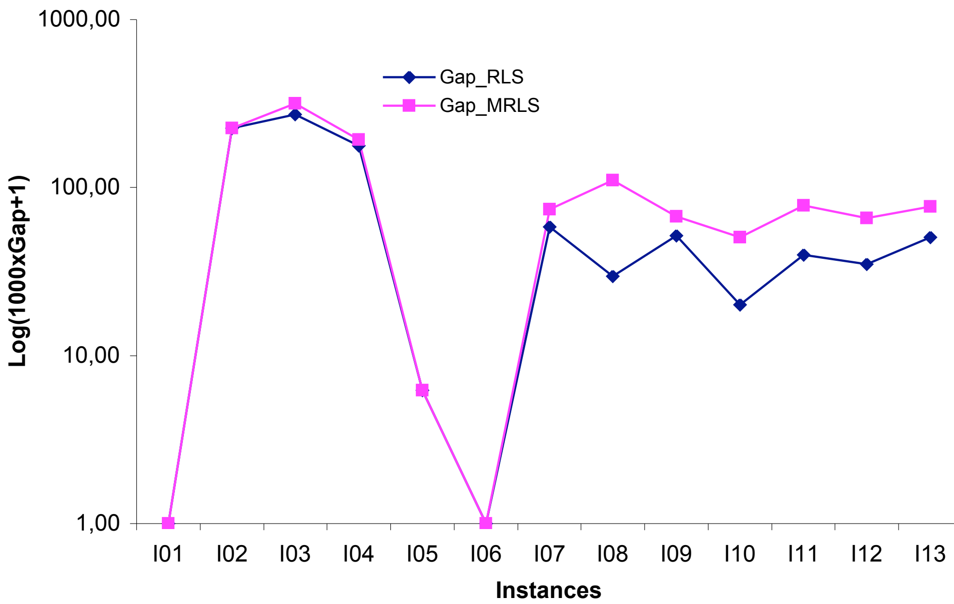

FIG. 4.12 – Comparaison des pourcentages d'amélioration de RLS et MRLS.

4.11 Conclusion

Nous avons traité dans ce chapitre le problème de sac-à-dos généralisé à choix multiple. Nous avons proposé deux algorithmes approchés.
Le premier algorithme s'appuie sur une "recherche guidée". Le principe de l'algorithme se résume en deux phases : (i) une première phase qui consiste à faire des interchanges pour tenter d'améliorer la solution en cours et (ii) une deuxième phase qui consiste à faire une diversification de la solution en acceptant une éventuelle dégradation de la solution en cours.
Le deuxième algorithme est composé de quatre phases : (i) une première phase qui consiste à partir d'une solution initiale et d'effectuer une recherche locale utilisant la notion d'inter-change, (ii) une deuxième phase qui est appliquée sur chaque solution estimée non-améliorante sur le voisinage en cours, (iii) une troisième phase qui consiste à effectuer une diversification et une quatrième phase qui consiste à introduire une mémoire afin d'éviter les recyclages sur certaines solutions explorées auparavant.
L'étude expérimentale montrait que le premier algorithme donnait des résultats meilleurs que les résultats de certaines approches de la littérature tout en consommant, en moyenne, le même temps d'exécution. Par ailleurs, le deuxième algorithme produisait des solutions de meilleure qualité (par rapport aux solutions réalisées par le premier algorithme) tout en nécessitant un temps d'exécution plus important.

Chapitre 5

Une méthode exacte pour le problème de sac-à-dos généralisé à choix multiple

Dans ce chapitre nous nous intéressons à l'étude du problème de sac-à-dos généralisé à choix multiple. Nous proposons une méthode de résolution exacte appliquant une procédure de séparation et évaluation. Le principe de la méthode s'appuie sur (i) la génération d'une solution de départ, (ii) la détermination d'une nouvelle borne supérieure utilisée aux différents niveaux de l'arborescence, et (iii) l'application de la stratégie par le meilleur d'abord.

5.1 Introduction

Souvent les algorithmes utilisés pour la résolution exacte des problèmes de type sac-à-dos s'appuie sur une procédure de séparation et évaluation. Cette dernière approche se base principalement sur le développement d'une arborescence composée d'un ensemble de nœuds dispersés sur différents niveaux. A chaque niveau, on fait correspondre une évaluation à l'aide de la fonction objectif représentant le problème à traiter. La solution obtenue sur un niveau donné est considérée comme une évaluation potentielle qui correspond à la meilleure solution en cours du problème.

Ce chapitre est dédié à l'étude du problème de sac-à-dos généralisé à choix multiple.

Ce problème peut aussi être considéré comme une variante du problème de sac-à-dos à choix multiple (Multiple-Choice Knapsack Problem : MCKP - voir Nauss [41]) et ce dernier est à son tour une variante du problème de sac-à-dos unidimensionnel ("knapsack Problem"). Nous évoquons le problème de sac-à-dos à choix multiple puisque ce dernier nous servira de point d'appui pour la construction des bornes supérieures pour le MMKP.
Dans un premier temps, nous commençons l'étude par la réduction du problème MMKP au problème MCKP, que l'on notera $MMKP_{aux}$ et qui sera considéré comme un problème auxiliaire pour le MMKP. Dans la même partie, nous présentons une borne supérieure ainsi qu'une borne inférieure de départ pour le MMKP.
Dans un deuxième temps, nous présentons une procédure de séparation et évaluation en s'appuyant sur une recherche par le meilleur d'abord. Finalement, nous présentons une étude expérimentale sur différents groupes d'instances de petite et moyenne tailles.

5.2 Les bornes supérieures et inférieures

Nous rappelons que le problème MMKP à variables binaires est défini par un ensemble de N éléments répartis sur un ensemble $\mathcal{J}$ de n classes $(J_i)_{(i=1,\ldots,n)}$ et disjointes entre elles et m contraintes de capacité, où à chaque objet $j \in J_i$ sont associés un profit v_{ij} pour $i = 1, \ldots, n$ et $j = 1, \ldots, r_i = |J_i|$, et un vecteur poids $W_{ij} = (w_{ij}^1, \ldots, w_{ij}^k, \ldots, w_{ij}^m)$.
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.
Le problème MMKP peur être écrit sous la forme suivante :

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

où x_{ij} désigne la variable de décision : si $x_{ij} = 1$ alors l'objet j de la classe i est sélectionné, et $x_{ij} = 0$ sinon. Pour ce problème, nous rappelons aussi les hypothèses suivantes (énoncées dans le chapitre 2) :

- Les valeurs de $v_{ij},\ w_{ij}^k$, pour $i = 1, \ldots, n,\ j = 1, \ldots, r_i$ et $k = 1, \ldots, m$ sont des entiers strictements positifs.
- Les valeurs C^k, pour $k = 1, \ldots, m$, sont des entiers strictement positifs.

Considérons le problème auxiliaire $MMKP_{aux}$ suivant :

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

où $C = \sum_{k=1}^{m} C^k$ et $w_{ij} = \sum_{k=1}^{m} w_{ij}^k$. On pose également $NR = \sum_{i=1}^{n} r_i - n$. On peut remarquer que le problème $MMKP_{aux}$ représente le problème MCKP.

Soit maintenant UB la valeur calculée à partir du problème modifié $MMKP_{aux}$ comme suit :

1. Pour chaque classe i, $i = 1, \ldots, n$ et pour chaque élément j tel que $j = 1, \ldots, r_i$, nous considérons l'élément ayant le plus grand rapport profit par poids $(\frac{v_{ij}}{w_{ij}})$. Nous notons cet élément par j_{max}^i pour chaque classe i.
2. Soient $R_{max} = \sum_{i=1}^{n} w_{ij_{max}^i}$ (réspectivement $V_{max} = \sum_{i=1}^{n} v_{ij_{max}^i}$) la valeur des poids cumulés (respectivement les profits cumulés) des éléments j_{max}^i de chaque classe i pour $i = 1, \ldots, n$. Deux cas se présentent :
   – $R_{max} > C$. Dans ce cas la valeur de UB est donnée par :
   $$UB = \sum_{i=1}^{n} v_{ij_{max}^i} \times (\frac{C}{\sum_{i=1}^{n} w_{ij_{max}^i}}) = V_{max} \times (\frac{C}{R_{max}}).$$
   – $R_{max} < C$. Dans ce cas, nous omettons pour chaque classe i l'élément j_{max}^i et nous trions les éléments restants et sans distinction de classes selon l'ordre

décroissant du rapport profit par poids. Les éléments sont cette fois-ci indexés par j, $j = 1, \ldots, NR$. Ainsi, l'élément j posséde un profit v_j et un poids w_j, pour $j = 1, \ldots, NR$.
L'ensemble de ces éléments forment un problème de sac-à-dos, KP, avec une capacité $C - R_{max}$. Nous calculons la borne supérieure UB_{KP} par la méthode de Dantzig [7]. Ce qui revient à remplir le sac par les éléments j, $j = 1, \ldots, NR$, jusqu'à sa saturation. L'élément critique ℓ pour $1 \leq \ell \leq NR$, correspondant à cette borne est donné par :

$$\ell = min\{j : \sum_{j=1}^{\ell-1} w_j \leq C - R_{max} < \sum_{j=1}^{\ell} w_j\}.$$

Ensuite nous calculons la borne UB_{KP} comme suit :

$$UB_{KP} = \sum_{j=1}^{\ell-1} v_j + (\frac{C - R_{max} - \sum_{j=1}^{\ell-1} w_j}{w_\ell}) \times v_\ell$$

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

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

Proposition 5.1 *UB est une borne supérieure pour le problème $MMKP_{aux}$.*

Preuve : Nous allons démontrer le premier cas ($R_{max} > C$) de cette proposition par l'absurde.
Tout d'abord rappelons la proprieté suivante :
Soient $(a, c, x, z,)$ des entiers positifs et (b, d, y, t) des entiers strictement positifs.

$$\text{Si } \frac{a}{b} \geq \frac{c}{d} \text{ et } \frac{x}{y} \geq \frac{z}{t} \quad (1)$$

$$\text{alors } \frac{a+x}{b+y} \geq \frac{c+z}{d+t} \quad (2).$$

Supposons qu'il existe une solution $\overline{X} = (\overline{x}_{1j_1}, \ldots, \overline{x}_{nj_n})$ pour le problème $MMKP_{aux}$ de valeur $\overline{V} = \sum_{i=1}^{n} v_{ij_i}$ telle que :

$$V_{max} \times (\frac{C}{R_{max}}) < \overline{V}.$$

$$\text{Donc} \quad \frac{V_{max}}{R_{max}} < \frac{\overline{V}}{C}.$$

$$\text{Or} \quad \frac{\overline{V}}{C} \leq \frac{\overline{V}}{\overline{\overline{R}}}, \text{ puisque } \overline{R} = \sum_{i=1}^{n} \overline{w}_{ij_i} \leq C.$$

$$\text{Donc} \quad \frac{V_{max}}{R_{max}} < \frac{\overline{V}}{C} \leq \frac{\overline{V}}{\overline{\overline{R}}}.$$

$$\text{La dernière double inégalité implique que} \quad \frac{V_{max}}{R_{max}} < \frac{\overline{V}}{\overline{\overline{R}}}.$$

D'après les relations $(1) - (2)$ et l'ordre décroissant des rapports des profits par poids des éléments :

$$\frac{\sum_{i=1}^{n} v_{ij^i_{max}}}{\sum_{i=1}^{n} w_{ij^i_{max}}} \geq \frac{\sum_{i=1}^{n} v_{ij_i}}{\sum_{i=1}^{n} w_{ij_i}}.$$

Nous avons alors $\frac{\overline{V}}{\overline{\overline{R}}}$ est le plus grand rapport ce qui est absurde.

Pour le deuxième cas $(R < C)$, il est facile de voir que les élements restants forment un problème de sac-à-dos et par la suite, la borne supérieure calculée est la borne de Dantzig [7].

□

Proposition 5.2 *UB est une borne supérieure pour le problème MMKP.*

Preuve : Soit $\overline{x}$ une solution optimale pour le MMKP, c'est-à-dire :

$$\begin{cases} Z_{MMKP}(\overline{x}) = \sum_{i=1}^{n}\sum_{j=1}^{r_i} v_{ij}\overline{x}_{ij} & \\ \text{tel que} \quad \sum_{i=1}^{n}\sum_{j=1}^{r_i} w_{ij}^k \overline{x}_{ij} \leq C^k, & k = 1, \ldots, m \\ \sum_{j=1}^{r_i} \overline{x}_{ij} = 1, & i = 1, \ldots, n \\ \overline{x}_{ij} \in \{0, 1\}. & \end{cases}$$

Puisque

$$\sum_{i=1}^{n}\sum_{j=1}^{r_i} w_{ij}^k \overline{x}_{ij} \leq C^k,\ \forall k=1,\ldots,m,\ alors\ \sum_{k=1}^{m}\sum_{i=1}^{n}\sum_{j=1}^{r_i} w_{ij}^k \overline{x}_{ij} \leq \sum_{k=1}^{m} C^k.$$

La dernière inégalité implique que :

$$\sum_{i=1}^{n}\sum_{j=1}^{r_i} w_{ij} \overline{x}_{ij} \leq C.$$

D'où $\overline{x}$ est une solution réalisable pour le problème modifié $MMKP_{aux}$.
Or $\forall x,\ Z_{MMKP_{aux}}(x) \leq UB$, par conséquent et en particulier pour $\overline{x}$, on déduit que $Z_{MMKP_{aux}}(\overline{x}) = Z_{MMKP}(\overline{x}) \leq UB$.

□

Par ailleurs, nous avons appliqué la deuxième version de l'algorithme (MRLS) proposé dans la partie B du chapitre 4 afin de detérminer une borne initiale (notée LB) pour le problème.

5.3 Algorithme de séparation et d'évaluation

Dans cette section nous proposons un algorithme de séparation et évaluation pour résoudre le problème MMKP.
Le principe de l'algorithme est de développer une arborescence de recherche permettant d'énumérer les solutions suivant l'ordre $\hat{X}_p, \hat{X}_{p-1}, \ldots, \hat{X}_{max}$ (d'évaluations respectives prises dans l'ordre $Z_p(\hat{X}_p) \geq Z_{p-1}(\hat{X}_{p-1}) \geq \ldots \geq Z_{max}(\hat{X}_{max})$) jusqu'à trouver la première solution réalisable.

5.3.1 Développement de l'arborescence de recherche

Chaque nœud développé dans l'arborescence correspond à un élément mis dans le sac. Une branche de l'arborescence correspond à une solution. Nous notons par n_{ij} le nœud développé correspondant à l'élément j de la classe i.
Les classes sont considérées une à une et pour chaque classe i, $i = 1, \ldots, n$, on trie les

éléments j, $j = 1, \ldots, r_i$, selon l'ordre décroissant de leurs profits respectifs, c'est-à-dire :

$$v_{i1} > v_{i2} > \ldots > v_{ij} > \ldots > v_{ir_i}.$$

Au cours du développement de l'arborescence et pour un nœud n_{ij}, nous développons deux nœuds :

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

Ceci est illustré par le schéma suivant :

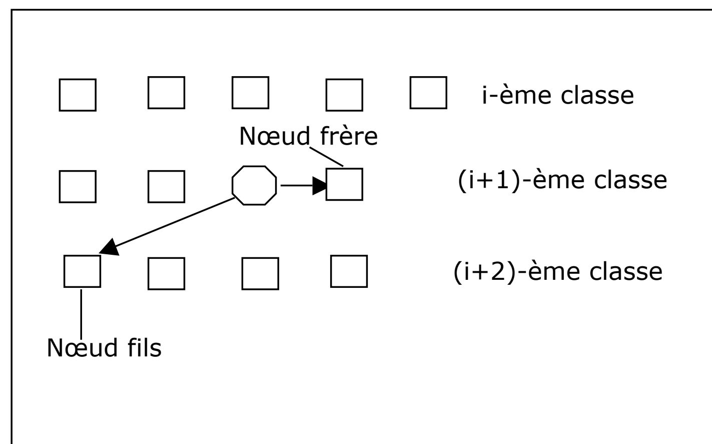

Fig. 5.1 – Schéma de développement de l'arborescence.

Le développement d'un nœud permet :

– d'une part, en développant le nœud frère, de générer une solution partielle ;
– d'autre part, en développant le nœud fils, de continuer le développement de la solution partielle courante.

Ceci est illustré par le schéma de la Figure 5.2.

L'approche de l'algorithme utilise la stratégie par *le meilleur d'abord*. Le choix du nœud à développer se fait suivant la valeur de la meilleure solution (réalisable ou non)

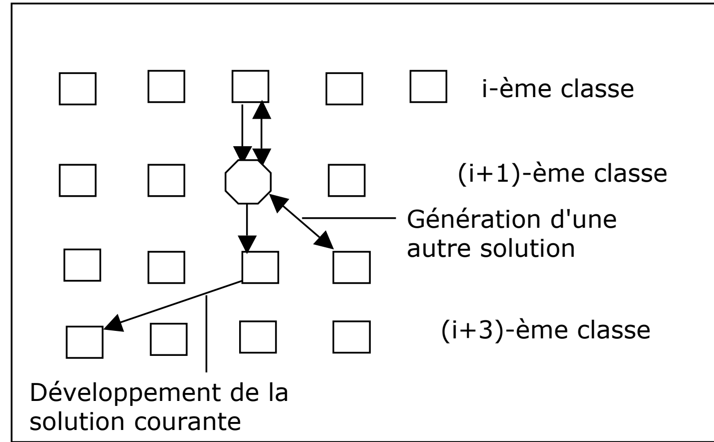

FIG. 5.2 – Schéma de développement d'un nœud.

qui pourra être obtenue à partir de ce nœud. La valeur de cette meilleure solution est facile à calculer. En effet, pour un nœud $n_{i,j}$ donné, la solution de plus grande valeur à partir de ce nœud est la solution obtenue en ajoutant les éléments $(i+1,1),(i+2,1),\ldots,(n,1)$, puisque tous les éléments de chaque classe ont été considérés selon l'ordre décroissant de leurs profits respectifs.

5.3.2 Principe de l'algorithme

Soient toutes les valeurs $Z_p(\hat{X}_p)$ des solutions $\hat{X}_p$ réalisables et non réalisables du problème du MMKP telles que :

$$Z_1(\hat{X}_1) \leq Z_2(\hat{X}_2) \leq \ldots \leq Z_l(\hat{X}_l) \leq \ldots \leq Z_p(\hat{X}_p).$$

Lemme 5.3 *Soit $Z_{max}(\hat{X}_{max})$ la plus grande valeur telle que $\hat{X}_{max}$ est réalisable, alors $\hat{X}_{max}$ est une solution optimale pour le MMKP.*
Si une telle solution n'existe pas (toutes les solutions $\hat{X}_l$ sont non réalisables, pour $1 \leq l \leq p$), alors le problème MMKP n'admet pas de solution.

Preuve : Si $\hat{X}_{max}$ est réalisable, alors $Z_{max}(\hat{X}_{max})$ est valeur d'une solution optimale puisqu'elle est la plus grande valeur obtenue pour une solution réalisable.

Nous aurons donc :

$$Z_{max}(\hat{X}_{max}) \leq Z_{max+1}(\hat{X}_{max+1}) \leq \ldots \leq Z_l(\hat{X}_l) \leq \ldots \leq Z_p(\hat{X}_p).$$

Avec $\hat{X}_{max+1},\ldots,\hat{X}_l,\ldots,\hat{X}_p$ sont des solutions non réalisables. □

Proposition 5.4 *L'algorithme de développement de l'arborescence ainsi défini développe toutes les solutions. De plus la première solution réalisable obtenue est optimale pour le MMKP.*

Preuve :

1. Pour démontrer que l'algorithme développe toutes les solutions, il suffit de démontrer que pour tout nœud $n_{i,j}$, l'algorithme permet de développer tous ses nœud fils.

   Lorsque le nœud $n_{i,j}$ est considéré, nous développons son nœud fils $n_{i+1,1}$. De la même façon, lorsque le nœud $n_{i+1,1}$ est traité, nous développons, en plus de son nœud fils, son nœud frère $n_{i+1,2}$. De même, quand le nœud $n_{i+1,2}$ est considéré, nous développons son nœud frère $n_{i+1,3}$ et ainsi de suite jusqu'à développer le dernier nœud $n_{i+1,r_{i+1}}$. L'algorithme permet donc de développer tous les nœuds fils $n_{i+1,1}, n_{i+1,2}, \ldots, n_{i+1,r_{i+1}}$ du nœud $n_{i,j}$.

2. Pour démontrer que la première solution réalisable trouvée est optimale pour le MMKP, il suffit de montrer que les solutions générées par l'algorithme sont obtenues dans l'ordre $\hat{X}_p, \hat{X}_{p-1}, \ldots, \hat{X}_{max}$ (d'évaluations respectives prises dans l'ordre $Z_p(\hat{X}_p) \geq Z_{p-1}(\hat{X}_{p-1}) \geq \ldots \geq Z_{max}(\hat{X}_{max})$). Ceci est vrai puisque l'algorithme considère les nœuds en meilleur d'abord suivant la meilleure solution obtenue par rapport à ce nœud.

□

5.3.3 Troncature des branches de l'arborescence

La troncature des branches de l'arborescence se fait grâce à des fonctions d'évaluation. A partir d'un nœud donné, deux types de troncatures peuvent être réalisés.

1. Un nœud fils développé à partir d'un nœud père tel que ce dernier correspond à une solution non réalisable est troncaturé (cf. Figure 5.3). Il faut remarquer qu'un nœud fils qui correspond à une solution non réalisable alors que le nœud père correspond à une solution réalisable ne doit pas être troncaturé (cf. Figure 5.4). De même qu'un nœud frère développé et qui correspond à une solution non réalisable ne doit pas non plus être troncaturé. En effet, la troncature de ces nœuds peut réduire l'espace de recherche des solutions réalisables puisque le nœud frère de ces nœuds peut correspondre à une solution réalisable.

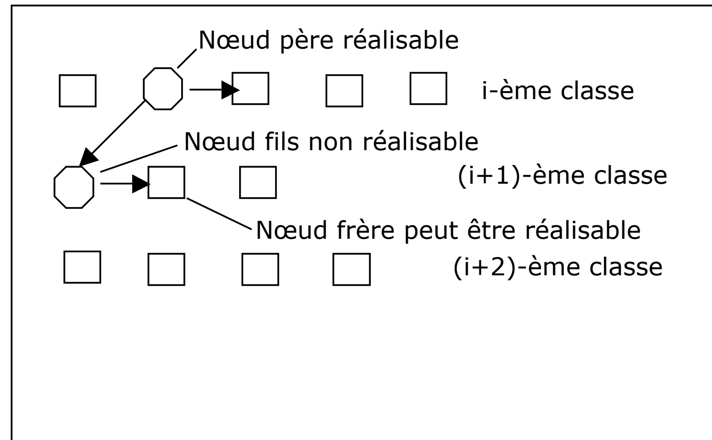

FIG. 5.3 – Premier schéma de troncature d'un nœud de l'arborescence.

2. Pour ce deuxième cas, nous avons besoin d'avoir en entrée la valeur d'une borne inférieure LB. Cette borne peut être celle présentée dans la partie B du chapitre 4. Avant le développement d'un nœud n_{ij} correspondant à une solution partielle de valeur $\hat{Z}$, nous calculons la borne supérieure UB décrite dans la section 5.2 pour le sous-problème MMKP constitué des classes J_p, pour $p = i+1, \ldots, n$.
Si $UB + \hat{Z} < LB$, nous sommes sûrs que toute solution réalisable développée à partir de ce nœud aura une valeur strictement inférieure à LB (solution réalisable) et donc elle n'est tout simplement pas optimale.

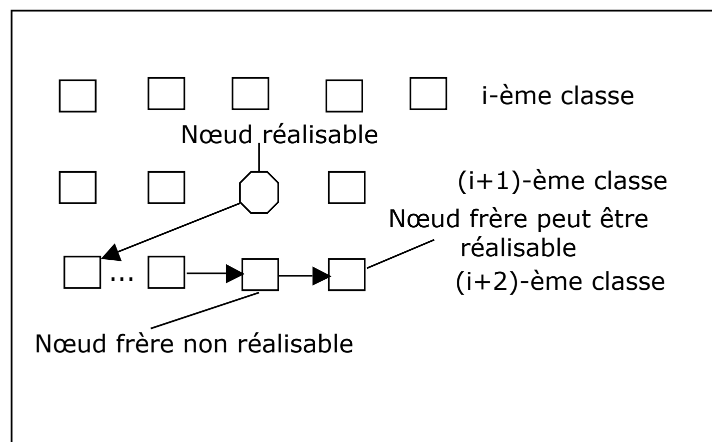

FIG. 5.4 – Deuxième schéma de troncature d'un nœud de l'arborescence.

5.3.4 L'algorithme

L'algorithme commence par le développement du nœud $n_{1,1}$ qui correspond au premier élément de la première classe. Il prend en entrée la valeur de la borne LB développée dans chapitre 4. Les éléments de chaque classe sont pris selon l'ordre décroissant de leurs profits respectifs.

L'algorithme termine dès qu'on obtient une solution réalisable qui est la solution optimale d'après la Proposition 5.4.

Les principales étapes de l'algorithme de séparation et évaluation pour le MMKP sont décrite dans la figure 5.5.

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

Initialisation :

1. LB ; /* une borne inférieure pour le MMKP */
2. Trier les éléments de chaque classe dans l'ordre décroissant des profits ;
3. $L = \{n_{11}\}$; /* nœud racine de l'arborescence */

Etape générale :

4. **Tant que** ($L \neq \emptyset$) **faire**
   - 4.1. nœud = meilleur nœud de L ;
   - 4.2. $L \longleftarrow L \setminus \{nœud\}$;
   - 4.3. Si nœud correspond à une solution réalisable, alors développer un nouveau nœud fils ;
   - 4.4. **Si** (*nœud fils* appartient à la dernière classe et la solution est réalisable) **alors** sortir avec la solution ;
   - 4.5. **Si** ($UB(nœud\ fils) > LB$) **alors** $L \longleftarrow L \cup \{nœud\ fils\}$;
   - 4.6. **Si** (*nœud* n'est pas le dernier élément d'une classe) **alors** Développer un *nœud frère* et le mettre dans L ;
5. **FinTant que**

FIG. 5.5 – Algorithme de résolution par séparation et évaluation pour le MMKP.

5.4 Partie expérimentale

D'une part, l'algorithme proposé résout à l'optimum toutes les instances de petite taille données par Khan *et al.* [28]. De plus, l'algorithme nécessitait un temps d'exécution "très faible" (trois secondes au maximum sur un SUN Ultra Sparc10) pour produire la solution optimale.

D'autre part, l'algorithme était incapable de résoudre les autres grandes instances (comme d'ailleurs l'algorithme développé par Khan *et al.* [28]). Cet echec est dû principalement à la place mémoire de la machine utilisée.

5.4.1 Génération des instances

Pour cela, nous avons testé l'algorithme sur un autre ensemble d'instances de petite et moyenne tailles. Ces instances ont été générées aléatoirement, réparties sur quatre groupes. Chacun des groupes représente les instances ayant le même nombre de classes. Pour ces différents groupes, le nombre d'éléments ainsi que le nombre de contraintes sont différents. Par ailleurs, nous générons une instance de chaque groupe comme suit : (i) fixer le nombre de classes, (ii) fixer le nombre d'éléments par classe, (iii) fixer le nombre de contraintes et (iv) génération aléatoire des profits dans l'intervalle $[0, 150]$ et des poids dans l'intervalle $[0, 50]$.

Le nombre de classes est fixé dans l'intervalle discret $\{10, 25, 50\}$, le nombre d'éléments par classe est fixé dans l'intervalle discret $\{5, 10, 15, 20\}$ et le nombre de contraintes par instance est fixé dans l'intervalle $\{5, 7, 10\}$. La capacité (C^k) de chaque contrainte de l'instance est posée égale à :

$$C^k = (1/2) \times (\sum_{i=1}^{n} \min_{1 \leq j \leq r_i} \left\{w_{ij}^k\right\} + \sum_{i=1}^{n} \max_{1 \leq j \leq r_i} \left\{w_{ij}^k\right\}),\ k = 1, \ldots, m.$$

La performance de l'algorithme proposé sur l'ensemble de ces instances est reportée dans les tableaux 5.1, 5.2, 5.3, et 5.4.

Pour chaque tableau, nous reportons :

- le nombre n de classes (colonne 2) ;
- le nombre r_i d'éléments dans chaque classe i, $i = 1, \ldots, n$ (colonne 3) ;
- le nombre m de contraintes (colonne 4) ;
- la borne inférieure LB (solution réalisable) obtenue en appliquant le deuxième algorithme du chapitre 4 (colonne 5) ;
- la borne supérieure UB proposée la proposition **??** (colonne 6)
- la solution optimale Z produite par l'algorithme (colonne 7) ;
- le temps d'exécution (noté T et mesuré en secondes : colonne 8) que nécessite l'algorithme pour donner la solution optimale.

Inst.	n : # Classes	r_i : # Items par classe	m : # Contraintes	LB	Opt	UB	T en secondes
I1a1	10	5	5	1420	1482	1558	0.1
I1a2	10	5	5	1392	1475	1513	0.1
I1a3	10	5	5	1375	1436	1562	0.1
I1a4	10	5	5	1387	1433	1520	0.1
I1a5	10	5	5	1442	1548	1598	0.1
I1b1	10	5	10	1559	1559	1602	0.1
I1b2	10	5	10	1553	1573	1623	0.1
I1b3	10	5	10	1461	1539	1569	0.1
I1b4	10	5	10	1049	1609	1635	0.1
I1b5	10	5	10	1371	1456	1502	0.1

TAB. 5.1 – Résultats numériques du premier groupe

Inst.	n : # Classes	r_i : # Items par classe	m : # Contraintes	LB	Opt	UB	T en secondes
I2a1	10	10	5	1662	1662	1662	0.3
I2a2	10	10	5	1649	1658	1679	0.8
I2a3	10	10	5	1592	1600	1623	1.2
I2a4	10	10	5	1426	1453	1493	0.9
I2a5	10	10	5	1461	1534	1582	0.5
I2b1	10	10	10	1665	1665	1682	0.4
I2b2	10	10	10	1655	1672	1723	0.5
I2b3	10	10	10	1629	1629	1723	0.2
I2b4	10	10	10	1598	1614	1722	0.5
I2b5	10	10	10	1591	1637	1682	0.4

TAB. 5.2 – Résultats numérique du second groupe

Inst.	n : # Classes	r_i : # Items par classe	m : # Contraintes	LB	Opt	UB	T en secondes
I3a1	25	10	5	4089	4137	4163	0.6
I3a2	25	10	5	4201	4245	4286	0.8
I3a3	25	10	5	4007	4043	4076	0.8
I3a4	25	10	5	4010	4045	4052	0.8
I3a5	25	10	5	4092	4123	4163	0.9
I3b1	25	10	10	4160	4177	4232	1.3
I3b2	25	10	10	4272	4278	4352	1.9
I3b3	25	10	10	3877	3982	4019	2.2
I3b4	25	10	10	4072	4105	4136	1.9
I3b5	25	10	10	4011	4071	4116	0.8

TAB. 5.3 – Résultats numérique du troisième groupe

5.4.2 Performance de l'algorithme exact

Dans un premier temps, nous nous sommes intéressés au comportement de l'algorithme en variant le nombre d'éléments par classe et le nombre de contraintes (les deux premiers groupes). Dans un deuxième temps, nous avons augmenté le nombre de classes. Finalement, nous avons considéré un groupe en augmentant le nombre de classes, en faisant une variation sur le nombre d'éléments par classe et en faisant une autre variation sur le nombre de contraintes. Ce dernier cas est représenté par le quatrième groupe pour lequel on souhaitait voir les limites (sur la machine utilisée) de l'algorithme proposé.

Les deux premiers groupes sont composés de dix instances chacun. Pour chacun des groupes, la moitié des instances comporte cinq contraintes et l'autre moitié dix contraintes. Le nombre de classes de l'ensemble de ces instances est fixé à dix et nous faisons une variation sur le nombre d'éléments par classe.

Nous avons considéré les instances des deux premiers groupes comme étant des instances de petite taille. Pour l'ensemble de ces instances, l'algorithme proposé convergeait vers la solution optimale en consommant un temps d'exécution inférieur à 1.2 secondes (le temps d'exécution maximum réalisé par l'algorithme sur l'ensemble des instances de ces groupes). A partir du tableau 5.1, nous pouvons remarquer que la variation sur le nombre de contraintes n'est pas signifcative pour ces instances, puisqu'il s'agit d'instances de petite taille (le nombre de variables n'augmente pas d'une façon significative). Après avoir augmenté le nombre d'éléments par classe, nous remarquons qu'à partir du tableau 5.2 que le temps d'exécution a augmenté sur l'ensemble des instances du deuxième groupe. Dans ce cas, on peut remarquer que le temps d'exécution est une fonction croissante du nombre d'éléments.

Pour le deuxième cas, nous avons considéré le troisième groupe composé de dix instances où le nombre de classes par instance a été augmenté. Cette augmentation permet aussi d'augmenter le nombre de variables par instance. Dans ce cas, nous pouvons constater qu'à partir du tableau 5.3 le temps d'exécution a pratiquement doublé.

Finalement, nous avons considéré le quatrième groupe (composé de quinze instances) dans lequel le nombre de classes a été fixé à 50 tout en effectuant des variation sur le nombre d'éléments par classe et sur le nombre de contraintes par instance. Notons que ce groupe comporte des instances ayant 1000 éléments répartis sur 50 classes à 20 éléments et soumis à 7 contraintes. A partir du tableau 5.4 on peut remarquer que

Inst.	n : # Classes	r_i : # Items par class	m : # Contraintes	LB	Opt	UB	T en secondes
I4a1	50	20	5	8514	8542	8569	7
I4a2	50	20	5	8561	8564	8589	6.5
I4a3	50	20	5	8619	8635	8676	8
I4a4	50	20	5	8443	8459	8489	7.3
I4a5	50	20	5	8432	8456	8482	15
I4b1	50	20	5	8575	8590	8623	12
I4b2	50	20	7	8561	8569	8587	9
I4b3	50	20	7	8545	8551	8586	10
I4b4	50	20	7	8526	8542	8583	11
I4b5	50	20	7	8438	8473	8529	13
I4c1	50	15	10	8289	8340	8369	22
I4c2	50	15	10	8303	8345	8372	20
I4c3	50	15	10	8472	8511	8568	35
I4c4	50	15	10	8529	8582	8613	28
I4c5	50	15	10	8323	8324	8392	18

TAB. 5.4 – Résultats numériques du quatrième groupe

le temps d'exécution croit avec le nombre de contraintes. De plus, nous constatons que malgré la diminution du nombre d'éléments par classe sur les cinq dernières instances, le temps d'exécution reste plus important à cause du nombre de contraintes imposées sur ces instances.

5.5 Conclusion

Dans ce chapitre, nous avons proposé un algorithme exact pour le problème du sac-à-dos généralisé à choix multiple. L'algorithme s'appuie sur une procédure de séparation et d'évaluation utilisant la stratégie du meilleur d'abord. Dans un premier temps, nous avons utilisé une borne inférieure de départ (solution réalisable) générée par l'application du deuxième algorithme du chapitre quatre. Dans un deuxième temps, nous avons effectué la réduction du problème initial sous forme d'un problème de sac-à-dos à choix multiple. Ce dernier problème nous a permis de calculer une borne supérieure pour le problème original ainsi que des bornes supérieures intermédiaires pour la méthode proposée. Par ailleurs, la combinaison entre la borne inférieure initiale et les bornes supérieures intermédiaires permettaient l'élagage d'un certain nombre de branches. L'étude expérimentale montrait que la méthode proposée était capable de résoudre des instances de petite et moyenne taille dont le nombre de variables pouvant comporter

1000 éléments répartis sur 50 classes à 20 éléments et soumis à 7 contraintes.

Chapitre 6

Conclusion et perspectives

6.1 Conclusion

Dans cette thèse, nous avons étudié deux variantes du probème de sac-à-dos. Plus précisément, nous avons traité le problème de la distribution équitable et le problème du sac-à-dos généralisé à choix multiple. La première partie est consacrée au développement d'algorithmes approchés pour les deux variantes du sac-à-dos. La deuxième partie traite essentiellement le problème du sac-à-dos généralisé à choix multiple. Nous avons proposé une méthode de résolution exacte qui s'appuie principalement sur : (i) le calcul de bornes et (ii) l'utilisation de la stratégie par le meilleur d'abord.
Le chapitre 3 porte sur l'étude du problème de la distribution équitable. Nous nous sommes intéressés à la résolution approchée de ce problème. Dans un premier temps, nous avons proposé une première version de l'algorithme exploitant certaines caractéristiques de la recherche tabou. Le principe de cette version de l'algorithme consiste : (i) à utiliser un paramètre profondeur limitant la recherche des solutions voisines autour des éléments dits "critiques" et (ii) à introduire une mémoire (liste tabou) permettant la non stagnation de certaines solutions. Dans un deuxième temps, nous avons proposé une deuxième version de l'algorithme. L'idée principale consiste à tenter la combinaison entre l'intensification de la recherche dans l'espace des solutions et la diversification de la solution. La stratégie adoptée portait principalement sur la modification de la façon d'exploiter le paramètre profondeur. Les deux versions de l'algorithme ont été testées sur un nombre d'instances de la littérature. Nous avons souligné la rapidité de

la première version et l'efficacité de la deuxième. L'étude expérimentale montrait que, par rapport à la première version de l'algorithme, la deuxième version produisait un nombre plus important de solutions optimales mais nécessitait un temps d'exécution plus important.

Le chapitre 4 traite le problème de sac-à-dos généralisé à choix multiple. Nous avons proposé deux algorithmes approchés.

Le premier algorithme s'appuie sur une "recherche guidée". Le principe de l'algorithme se résume en deux phases : (i) une première phase qui consiste à faire des interchanges pour tenter d'améliorer la solution en cours et (ii) une deuxième phase qui consiste à faire une diversification de la solution en acceptant une éventuelle dégradation de la solution en cours.

Le deuxième algorithme est composé de quatre phases : (i) une première phase qui consiste à partir d'une solution initiale et d'effectuer une recherche locale utilisant la notion d'interchange, (ii) une deuxième phase qui est appliquée sur chaque solution estimée non-améliorante sur le voisinage en cours, (iii) une troisième phase qui consiste à effectuer une diversification et une quatrième phase qui consiste à introduire une mémoire afin d'éviter les recyclages sur certaines solutions explorées auparavant.

L'étude expérimentale montrait que le premier algorithme donnait des résultats meilleurs que les résultats de certaines approches de la littérature tout enconsommant, en moyenne, le même temps d'exécution. Par ailleurs, le deuxième algorithme produisait des solutions de meilleure qualité (par rapport aux solutions réalisées par le premier algorithme) tout en nécessitant un temps d'exécution plus important.

Finalement, nous avons proposé dans le chapitre 5 une méthode de résolution exacte pour le problème du sac-à-dos généralisé à choix multiple. Dans un premier temps, nous avons proposé la réduction du problème initial au problème de sac-à-dos à choix multiple. Ce dernier problème représente un problème auxiliaire pour le problème initial. Par la suite, nous avons proposé une borne supérieure ainsi qu'une borne inférieure de départ pour le problème. Finalement, nous avons proposé les étapes principales de l'algorithme en s'appuyant sur une procédure de séparation et évaluation, utilisant une recherche par le meilleur d'abord. L'étude expérimentale montrait l'efficacité de la méthode proposée sur différents groupes d'instances de petite et moyenne taille.

6.2 Perspectives

Cette section est composée de trois parties. Dans la première partie nous proposons une étude préliminaire sur la réduction de la taille du problème de sac-à-dos généralisé à choix multiple. Dans la deuxième partie, nous nous intéressons à la généralisation de l'aspect fixation des variables (et de classes) à l'optimum à tous les niveaux de l'arborescence développée par l'hybridation d'une méthode de séparation et évaluation ainsi qu'une relaxation surrogate. Dans la denière partie, nous intéressons aux méthodes évolutionistes récentes.

6.2.1 Fixation des variables pour le sac-à-dos généralisé à choix multiple

Cette section présente la notion de fixation de variables qui peut être intégrée comme une phase préliminaire dans une méthode de résolution exacte. Nous nous intéressons au problème de sac-à-dos généralisé à choix multiple. En particulier, cette réduction peut être opérée en phase de prétraitement de la méthode de séparation et évaluation développée dans le chapitre 5. Cette réduction s'appuie principalement sur le résultat de Fayard et Plateau [9] adapté au problème de sac-à-dos généralisé à choix multiple. Le résultat suivant traite les cas où les variables peuvent être fixées à l'optimum.

Proposition 6.1 *Soit* $\underline{X} = \{x_{ij}\},\ i = 1;\ldots,n,\ \ j = 1,\ldots,r_i$, *une solution réalisable pour le MMKP. Pour toute classe* J_i, $i = 1,\ldots,n$, *pour tout élément* $j \in \{1,\ldots,r_i\}$, *et pour tout* $x_{i_0 j_{i_0}} = \epsilon$ *dans* $\underline{X}$, *si on a :*

$$Z(MMKP|x_{i_0 j_{i_0}} = 1-\epsilon) < Z(\underline{X}),$$

alors $x_{i_0 j_{i_0}}$ *prend la valeur* ϵ *dans toute solution optimale de MMKP.*

Preuve : Soit J_{i_0} une classe donnée et soit j_{i_0} un élément de la classe J_{i_0} tels que l'on a $Z(MMKP|x_{i_0 j_{i_0}} = 1-\epsilon) < Z(\underline{X})$.

Supposons qu'il existe un vecteur solution $\underline{X} = \{x_{ij}\}$, pour $i = 1,\ldots,n$, $j = 1,\ldots,r_i$ tel que l'optimum soit atteint pour l'élément j_{i_0} de la classe J_{i_0}.

$$Z(MMKP) = Z(MMKP|x_{i_0 j_{i_0}} = 1-\epsilon),$$

$$Z(\underline{X}) \leq Z(MMKP) = Z(MMKP|x_{i_0 j_{i_0}} = 1 - \epsilon).$$

Contradiction avec l'hypothèse $Z(MMKP|x_{i_0 j_{i_0}} = 1 - \epsilon) < Z(\underline{X})$.
Donc la valeur optimale pour le MMKP ne peut être atteinte pour $x_{i_0 j_{i_0}} = 1 - \epsilon$.
Dans ce cas, l'élément $x_{i_0 j_{i_0}}$ prend la valeur ϵ dans toute solution optimale. □

Etant donné que nous ne disposons pas de la solution optimale du problème, il est possible de se contenter d'une borne supérieure et de remplacer la valeur exacte $Z(MMKP|x_{ij} = 1 - \epsilon)$ dans l'inégalité de la proposition ci-dessus par une borne supérieure $UB(MMKP|x_{ij} = 1 - \epsilon)$.
Notons que le nombre de variables fixées à l'optimum dépend, en général, de la qualité de la borne supérieure et de la solution réalisable de départ.
Cette proposition permet, comme il sera développé dans la suite, de fixer aussi bien des variables à l'intérieur des classes que de fixer des classes entières.

A) Fixation des variables à l'intérieur des classes

Afin de fixer des variables à l'intérieur des classes, nous considérons les variables x_{ij} d'une solution réalisable ayant la valeur 1. Ainsi, si la Propositioné 6.1 est vérifiée pour une telle variable $x_{ij} = 1$, cette variable aura la valeur 0 dans toute solution optimale.

B) Fixation d'une classe

Dans ce cas, nous considérons cette fois-ci les variables x_{ij} d'une solution réalisable ayant la valeur 0. Si la Proposition 6.1 est vérifiée pour une variable $x_{ij} = 0$ dans une classe J_i, cette variable aura la valeur 1 dans toute solution optimale. En effet, étant donnée que la contrainte

$$\sum_{j=1}^{r_i} x_{ij} = 1, \; i = 1, \ldots, n,$$

exprime le fait, qu'à l'optimum un seul élément d'une classe est fixé à 1, alors on est sûr que tous les autres éléments de la classe J_i seront fixés à 0. Ceci permet de fixer toute la classe et de réduire par la même occasion le problème de départ de toute la classe.

6.2.2 Relaxation lagrangienne et montée surrogate

Dans cette section, nous décrivons l'idée principale d'une recherche future pour la résolution exacte du problème de sac-à-dos généralisé à choix multiple. Nous supposons aussi que nous disposons d'une méthode de recherche s'appuyant sur une procédure de séparation et d'évaluation. L'idée revient donc à répéter les deux étapes suivantes :

1. Réduire le (sous)problème en appliquant une fixation de variables (ou de classes).
2. Effectuer une séparation au niveau du sommet en cours (ou choisi) si nécessaire.

Initialement, nous partons du problème original, nous effectuons une fixation de variables (et de classes), puis nous appliquons une procédure de séparation et évaluation sur le problème réduit. Maintenant supposons que l'on se trouve sur un nœud interne de l'arborescence développée. Dans ce cas, le nœud correspond à un sous-problème et donc nous pouvons appliquer les étape (1) et (2) sur ce sous-problème. Nous pensons que l'application de ce processus peut être réalisé en considérant une des approches suivantes : (i) relaxation lagrangienne et (ii) la montée surrogate.
Pour le sac-à-dos généralisé à choix multiple, nous pouvons voir que la relaxation lagrangienne permet de donner le problème suivant :

$$L(MMKP)\begin{cases} \text{Maximiser} & \hat{Z}(x) = \Big\{ \sum_{i=1}^{n} \sum_{j=1}^{r_i} (v_{ij} - \sum_{k=1}^{m} \lambda^k w_{ij}^k) x_{ij} + \sum_{k=1}^{m} \lambda^k C^k \Big\} \\ \text{s.c} & \sum_{j=1}^{r_i} x_{ij} \leq 1, \quad i = 1, \ldots, n \\ & 0 \leq x_{ij} \leq 1, \quad j = 1, \ldots, r_i. \end{cases}$$

On peut remarquer que l'évaluation du problème $L(MMKP)$ représente une borne supérieure pour le problème original. Dans ce cas, l'idée de la fixation des variables x_{ij}, pour $i = 1, \ldots, n \jmath = 1, \ldots, r_i$ peut s'effectuer, par exemple, en résolvant des sous-gradients comme suit :

$$\bar{x}_{ij} = \begin{cases} 1 & \text{si } (v_{ij} - \sum_{k=1}^{m} \lambda^k w_{ij}^k) > 0 \ , \\ 0 & \text{sinon.} \end{cases}$$

Notons que toute la difficulté pour une meilleure fixation des varaibles demeure dans la mise à jour des multiplicateurs de Lagrange.
De la même façon, nous pouvons appliquer le même procédé tout en utilisant la montée

surrogate. Cette approche consiste à faire la somme de toutes les contraintes pour n'obtenir qu'une seule et d'opérer ensuite une relaxation lagrangienne sur le problème obtenu. Dans notre cas, la relaxation surrogate est obtenue comme suit :

$$SL(MMKP)\begin{cases} \text{Maximiser} & \bar{Z}(x) = \sum_{i=1}^{n}\sum_{j=1}^{r_i} v_{ij}x_{ij} & \\ \text{s.c} & \sum_{k=1}^{m}\pi^k \sum_{i=1}^{n}\sum_{j=1}^{r_i} w_{ij}^k x_{ij} \leq \sum_{k=1}^{m}\pi^k C^k, & k = 1,\dots,m \\ & \sum_{j=1}^{r_i} x_{ij} \leq 1, & i = 1,\dots,n \\ & 0 \leq x_{ij} \leq 1, & i = 1,\dots,n, \jmath = 1,\dots,r_i. \end{cases}$$

De la même façon, nous pensons qu'un bon choix des multiplicateurs surrogate $\pi^k,\ k =\ 1,\dots,m$, permettrait une meilleure fixation des variables.

6.2.3 Méthodes évolutionnistes

La méthode génétique

Souvent les "bonnes" solutions ou solutions efficaces saturent au moins une contrainte. Nous qualifions ce type de solutions de *Configurations Pertinentes*. Les heuristiques qui font appel aux techniques d'inter-échanges ou *swapping* afin d'améliorer la meilleure solution en cours, et aux stratégies de diversification telles que la dégradation, l'historique (ou mémoire) des solutions, le démarrage à partir d'un autre point trouvent leurs limites. Plus précisément, lorsqu'il s'agit d'une configuration pertinente, ll est très difficile de l'améliorer en s'appuyant sur la notion d'inter-échange d'un ou plusieurs éléments. Nous qualifions un tel état "d'*état de stagnation*". Dans une telle situation, les heuristiques font appel aux techniques de diversification permettant de contourner le problème en identifiant un autre espace d'exploration. Ce type d'opération garantit une exploration transversale large, et non pas une exploration en profondeur. Car une telle approche n'a pas les outils nécessaires pour améliorer ou construire des solutions à partir des Configurations Pertinentes (bouclage ou confinement autour d'un optimum local).

Notre réflexion de base s'appuie sur la construction des solutions de "bonne" qualité à partir des Configurations Pertinentes. Notre approche prend en compte deux

hypothèses. La première est qu'une solution de "bonne" qualité possède un voisinage dont les solutions ont aussi de "bonne" qualité. La sélection d'une configuration pourra se faire à partir d'une classe de Configurations Pertinentes. Nous appliquons ensuite les étapes générales d'un algorithme génétique (génération de la population initiale, évaluation fonction fitness, sélection, reproduction et remplacement, mutation,...).

La méthode de Path-Relinking

Le processus de produire des combinaisons linéaires de solutions de référence peut être vu comme étant produire des chemins entre des solutions qui mènent à des points entre plusieurs solutions. Un chemin construit entre des solutions dans un espace voisinage produit des solutions qui partagent une série d'attributs avec les solutions originales. Ces attributs peuvent être des nœuds, des valeurs de variables, etc.. L'idée de path relinking est de chercher des solutions qui partagent des attributs avec d'anciennes solutions avec l'espoir d'obtenir de meilleures solutions. Path relinking construit de nouvelles solutions en exploitant les trajectoires qui conduisent aux bonnes solutions, en commençant par certaines solutions, appelée la *solution d'initiation* (initiating solution), et en produisant un chemin dans le voisinage de l'espace qui conduit à d'autres solutions, appelées les *solutions de guidage* (guiding solutions). Souvent on introduit la notion de recherche Tabou pour pouvoir construire ces chemins sans les répéter à l'étape suivante.

Nous avons construit pour le porblème du KSP des chemins considérant la combinaison d'attributs de plusieurs solutions (multiparent path) et les rendant plus "fines"(tunneling) en essayant de combiner deux solutions dans des voisinages différents. Nous avons utilisé path relinking pour améliorer les procédures de réinitialisation. Dans notre procédure constructive CP, la construction d'une solution est indépendante de la construction de la solution suivante. On ne garde pas une histoire. On a proposé deux applications possibles de path relinking en combinaison avec la procédure CP :

- utiliser path relinking comme un processus de post-optimisation à toutes les paires de solutions "élites" trouvées avec la procédure constructive CP,
- utiliser path relinking comme une stratégie d'intensification appliquée à chaque solution obtenue après la phase de recherche locale.

Les deux stratégies gardent le plus petit ensemble de solutions "élites" choisies après chaque processus de recherche locale. Appliquer path relinking après chaque recherche

locale pour KSP donne, en général, de meilleurs résultats. Dans ce cas on prend la solution de recherche locale $\underline{X}$ et une solution choisie aléatoirement dans l'ensemble des solutions "élites" $\overline{X}$. On obtient les ensembles de mouvements qui peuvent être appliqués pour atteindre la solution de guidage à partir d'une solution initiale. On prend la meilleure solution dans cette trajectoire et on la considère comme solution candidat à ajouter aux solutions "élites". Finalement, on pense pouvoir paralléliser facilement ces procédures pour augmenter leur efficience.

Bibliographie

[1] E. Balas and E. Zemel, *An algorithm for large zero-one knapsack problem*, Operations Research, vol. 28, pp.1130-1154, 1980.

[2] J.-P. Barthélemy, G. Cohen et A. Lobestein, *Complexité Algorithmique et Problèmes de Communications*, Collection CNET-ENST, Paris, 1992.

[3] J.R. Brown, *The knapsack sharing*, Operations Research, vol. 27, pp. 341-355, 1979.

[4] J.R. Brown, *Solving knapsack sharing with general tradeoff functions*, Mathematical Programming, vol. 51, pp.55-73, 1991.

[5] P. Chu and J.E. Beasley, *A genetic algorithm for the set covering problem,* European Journal of Operational Research, vol.94, pp.392-404, 1996.

[6] P. Chu and J.E. Beasley, *A genetic algorithm for the multidimensional knapsack problem*, European Journal of Operational Research, vol. 123, pp. 333-345, 2000.

[7] G. B. Dantzig, *Discrete variable extremum problems*, Operations Research, vol. 5, pp. 266-277, 1957.

[8] K. Dudinski and S. Walukiewicz, *Exact methods for the knapsack problem and its generalizations*, Mathematical Programming, vol. 29, pp. 231-249, 1984.

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

[10] A. Freville and G. Plateau, *The 0-1 bidimensional knapsack problem : toward an efficient high-level primitive tool*, Journal of Heuristics, vol. 2, pp. 147-167, 1997.

[11] M. Garey and D. Johnson, *Computers and Intractability : a Guide to the Theory of NP-Completness*, W. H. Freeman and Company, San Francisco, USA, 1979.

[12] P.C. Gilmore and R.E. Gomory, *A Linear programming approach to the cutting stock problems*, Operations Research, vol. 9, 1961.

[13] P.C. Gilmore and R.E. Gomory, *The theory and computation of knapsack functions*, Operations Research, vol. 13, pp. 879-919, 1966.

[14] F. Glover, *Future paths for integer programming and links to artificial intelligence*, Computers and Operations Research, vol. 13, pp. 533-549, 1986.

[15] F. Glover and M. Laguna, *Tabu search*, Kluwer Academic Publishers, 1997.

[16] P. Hansen, *The steepest ascent mildest descent heuristic for combinatorial programming*, Presented at the Congress on Numerical Methods in Combinatorial Optimization, Capri, Italy, 1986.

[17] S. Haykin, *Neural Networks*, Macmillan, 1994.

[18] M. Hifi, *Exact algorithms for large-scale unconstrained two and three staged cutting problems*, Computational Optimization and Applications, vol. 18, pp. 63-88, 2001.

[19] M. Hifi, M. Michrafy and A. Sbihi, *Algorithms for the multiple-choice multidimensional knapsack problem*, working paper, CERMSEM, Université Paris 1, Les Cahiers de la M.S.E, vol. 31, 2002. (Presented in seminary Modélisation et Résolution Algorithmique), 2003.

[20] M. Hifi, M. Michrafy and A. Sbihi, *A reactive local search based-Algorithm for the multiple-choice multidimensional knapsack problem*, Les Cahiers de la M.S.E, vol. 33, 2003.

[21] M. Hifi, C. Roucairol, *Approximate and exact algorithms for constrained (un)weighted two-dimensional two-staged cutting stock problems*, Journal of Combinatorial Optimization, 2001, vol.5, pp.465-494.

[22] M. Hifi and S. Sadfi, *The knapsack sharing problem : an exact algorithm*, Journal of Combinatorial Optimization, vol.6, pp.35-45, 2002.

[23] M. Hifi, S. Sadfi and A. Sbihi, *Efficient algorithms for the knapsack sharing problem*, Les Cahier de la MSE, vol.122, 2000.

[24] M. Hifi, S. Sadfi and A. Sbihi, *An efficient algorithm for the knapsack sharing problem*, Computational Optimization and Applications, vol.23, pp. 27-45, 2002.

[25] M. Hifi, V. Zissimopoulos, *A recursive exact algorithm for weighted two-dimensional cutting*, European Journal of Operational Research, vol.91, pp.553-564, 1996.

[26] E. Horowitz and S. Sahni, *Computing partitions with applications to the knapsack problem*, Journal of ACM, vol. 21, pp. 277-292, 1974.

[27] 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, vol. 2/1, pp. 154-174, 2002.

[28] 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, May 1998.

[29] P. Kilby, P. Prosser and P. Shaw, *Guided local search for the vehicle routing problem*, Proceeding of the 2nd International Conference on Mataheuristics (MIC97), Sophia-Antipolis, France, pp. 21-24, 1997.

[30] T. Kuno, H. Konno and E. Zemel, *A linear-time algorithm for solving continuous maximum knapsack problems*, Operations Research Letters, vol. 10, pp. 23-26, 1991.

[31] H. Luss, *Minmax resource allocation problems : optimization and parametric analysis*, European Journal of Operational Research, vol. 60, pp. 76-86, 1992.

[32] M. Magazine and M. Chern, *A note on approximation schemes for multidimensional knapsack*, Mathematics of Operations Research, vol. 9, pp. 244-247, 1984.

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

[34] S. Martello, D. Pisinger and P. Toth, *Dynamic programming and strong bounds for the 0-1 knapsack problem*, Management Science, vol. 45, pp. 414-424, 1999.

[35] S. Martello, P. Toth, *Upper bounds and algorithms for hard 0-1 knapsack problems*, Operations Research, vol.45, pp.768-778, 1997.

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

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

[38] S. Martello and P. Toth, *Algorithms for knapsack problems*, Annals of Discrete Mathematics, vol.31, pp. 70-79, 1987.

[39] R. Morabito, M. Arenales, *Performance of two heuristics for solving large scale two-dimensional guillotine cutting problems*, INFOR, vol. 33, pp. 145-155, 1995.

[40] M. Moser, D. P. Jokanović and N. Shiratori, *An algorithm for the multidimesional multiple-choice knapsack problem*, IEICE Transactions on Fundamentals of Electronics, vol. 80, No 3, pp. 582-589, 1997.

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

[42] J.S. Pang and C.S. Yu, *A min-max resource allocation problem with substitutions*, European Journal of Operational Research, vol. 41, pp. 218-223, 1989.

[43] D. Pisinger, *An exact algorithm for large multiple knapsack problems*, European Journal of Operational Research, vol. 114, pp. 528-541, 1999.

[44] D. Pisinger, *Core problems in knapsack algorithms*, Operations Research, vol. 47, pp. 570-575, 1999.

[45] D. Pisinger, *A minimal algorithm for the 0-1 knapsack problem*, Operations Research, vol. 45, pp.758-767, 1997.

[46] D. Pisinger, *An expanding-core algorithm for the exact 0-1 knapsack problem*, European Journal of Operational Research, vol. 87, pp. 175-187, 1995.

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

[48] S. Sadfi, *Méthodes adaptatives et méthodes exactes pour le problème du knapsack linéaire et non linéaire*, Thèse de doctorat en informatique, Université Paris 11 Paris Sud Orsay, février 1999.

[49] A. Sbihi et M. Hifi, *Algorithmes de recherche locale réactive pour le problème du Sac-à-Dos Multicontraint à Choix Multiple*, FRANCORO IV - Conférence Internationale en Recherche Opérationnelle, Université de Fribourg et École d'ingénieurs du canton de Vaud, Fribourg-Suisse, du 18 au 21 août 2004.

[50] A. Sbihi, M. Hifi and M. Michrafy, *A Reactive Local Search-Based Algorithm for a variant of the Knapsack Problem*, INOC'2003 - International Network Optimization Conference, INT, Evry/Paris-France, du 27 au 29 octobre 2003.

[51] A. Sbihi and M. Hifi, *A Guided Local Search-Based Algorithm for the Multiple-choice Multidimensional Knapsack*, ICOR'03 - 6th International Conference of Operations Research, La Havane-Cuba, du 15 au 19 septembre 2003.

[52] A. Sbihi, M. Hifi et S. Sadfi, *Un Algorithme Approché pour le Problème de Knapsack Sharing*, CIRO'02 - III-ème Conférence Internationale en Recherche Opérationnelle, UCAM, Marrakech-Maroc, du 4 au 6 juin 2002.

[53] A. Sbihi, M. Hifi et S. Sadfi, *Une recherche tabou pour le problème de la distribution équitable*, Quatrièmes journées de la ROADEF, ENST-Paris, du 20 au 22 février 2002.

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

[55] C.S. Tang, *A max-min allocation problem : its solutions and applications*, Operations Research, vol. 36, pp. 359-367, 1988.

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

[57] E.P.K. Tsang and C. Voudouris, *Fast local search and guided local search and their application to British Telecom's worforce scheduling problem*, Operations Research Letters, vol. 20, No.3, pp. 119-127, 1997.

[58] J.M. Valéro de Carvalho, AJ. Ridrigues, *An LP-based approach to a two-stage cutting stock problem*, European Journal of Operational Research, vol.84, pp.580-589, 1995.

[59] C. Voudouris and E.P.K. Tsang, *Guided local search*, Technical Report CMS-247, Department of Computer Science, University of Essex, August 1995.

[60] C. Voudouris and E.P.K. Tsang, *Partial constraint satisfaction problems and guided local search for combinatorial*, *in* Proceedings of Practical Application of Constraint Technology (PACT'96), pp. 337-356, 1996.

[61] C. Voudouris and E.P.K. Tsang, *Guided local search and its application to the travelling salesman problem*, European Journal of Operational Research, vol.113, pp. 469-499, 1999.

[62] T. Yamada and M. Futakawa, *Heuristic and reduction algorithms for the knapsack sharing problem*, Computers and Operations Research, vol. 24, p. 961-967, 1997.

[63] T. Yamada, M. Futakawa and S. Kataoka, *Some exact algorithms for the knapsack sharing problem*, European Journal of Operational Research, vol. 106, 177-183, 1998.

Résumé

La thèse se situe dans le domaine de l'optimisation combinatoire, en particulier celui de la modélisation et de la résolution algorithmique. Dans cette thèse, nous étudions deux variantes NP-difficiles de problème de type sac-à-dos. Plus précisément, nous traitons le problème de la distribution équitable (le Knapsack Sharing Problem : KSP) et le problème du sac-à-dos généralisé à choix multiple (le Multiple-choice Multidimensional Knapasck Problem : MMKP). Dans la première partie de cette thèse, nous nous intéressons au développement d'algorithmes approchés pour les deux variantes évoquées du problème de type sac-à-dos. La deuxième partie traite essentiellement de la résolution exacte du problème du sac-à-dos généralisé à choix multiple. L'approche exacte que nous proposons est de type séparation et évaluation s'appuyant principalement sur : (i) le calcul des bornes inférieure et supérieure et (ii) l'utilisation de la stratégie par le meilleur d'abord en développant des branches à double noeuds fils et frère.

La première partie porte sur l'étude et la résolution approchée des deux problèmes KSP et MMKP. Concernant le problème de la distribution équitable, nous proposons dans un premier temps, une première version de l'algorithme exploitant certaines caractéristiques de la recherche tabou. Dans un deuxième temps, nous développons une deuxième version de l'algorithme dont l'idée principale consiste à tenter de combiner l'intensification de la recherche dans l'espace des solutions et la diversification de la solution obtenue. Nous soulignons la rapidité de la première version et l'efficacité de la deuxième. Ensuite nous nous intéressons au problème de sac-à-dos généralisé à choix multiple. Nous proposons deux heuristiques de recherche locale itérative. Le premier algorithme s'appuie sur une "recherche guidée". Le deuxième algorithme est une recherche locale que nous appelons réactive avec stratégies de déblocage et de dégradtion améliorantes de la solution et basées sur l'inter-change local.

Dans la deuxième partie de cette thèse, nous proposons une méthode de résolution exacte de type séparation et évaluation pour le problème du sac-à-dos généralisé à choix multiple. D'une part, nous nous proposons la réduction du problème initial au problème auxiliaire MMKP$_{aux}$ qui n'est autre que le problème de sac-à-dos à choix multiple MCKP. Nous calculons une borne supérieure pour le MMKP$_{aux}$ et nous établissons le résultat théorique pour lequel une borne supérieure pour le MMKP$_{aux}$ est une borne supérieure pour le MMKP. D'autre part, nous proposons le calcul d'une borne supérieure ainsi qu'une borne inférieure de départ pour le problème étudié qui sont nécessaires pour la réduction de l'espace de recherche. L'étude expérimentale montre l'efficacité de la méthode proposée sur différents groupes d'instances de petite et moyenne taille. Nous expliquons enfin pourquoi cet algorithme exact atteint ses limites de résolution, dûes principalement à la complexité intrinsèque du modèle étudié. D'autant la résolution dépend de la taille et la densité des instances traitées.

Mots clés

Optimisation combinatoire, Sac-à-dos, Heuristique, Recherche tabou, Méthodes exactes.

Abstract

The thesis is in the field of the particular field of operations research and combinatorial optimisation that is the modeling and algorithmic resolution. In this thesis we propose to study two particular NP-Hard variant problems of the binary knapsack type. More precisely, we treat the Knapsack Sharing Problem namely KSP and the Mutiple Choice Multidimensional Knapsack Problem namely MMKP. The first part of this dissertation is concerning to develop approxiamate algorithms for the two evoked variants of the 01 knapsask problem. The second part is particularly concerning with the exact resolution of the MMKP. The exact approach which we propose is of branch-and bound type which : (i) computes of lower and upper bounds and (ii) develops a son-brother double node branches with a best first startegy of exploration.

Indeed, the first part gets the study of the two problems KSP and MMKP. We are interested to develop approximate algorithms based upon local search strategies. First and concerning the KSP, we propose a first version of an approximate algorithm based upon tabu search strategy. Second, we enhance this version by combining the intensification of the search in the space of solutions and the diversifiaction of the obtained solution. Experimental results shows the fastness ofthe first version and the effectiveness and the efficiency of the second one. Next, we propose two iterative local search methods for the MMKP. The first one is an heuristic based guided local search end the second is a heuristic that we call a reactive local search approach with some improving degrading and debloking strategies of the solution based local swapping search.

In the second part of the dissertation, we propose an optimal method based branch-and-bound for solving the MMKP. First, we propose to transform the MMKP into an MMKP$_{aux}$ problem which is a Multiple-Choice Knapsack Problem MCKP. We compute an upper bound for MMKP$_{aux}$ and we establish the theoritical result for which an upper bound for MMKP$_{aux}$ is also an upper bound for MMKP. After, we deal with the computing of lower and upper bounds which are necessary to reduce the space search in a branch-and-bound approach. Add to this, the resolution strongly depends on the density and the size of the treated instances. The experimental study shows the the efficiency of the proposed algorithm which is able tos solve different groups of instances of small and medium sizes. Finally, we explain the limits of the branch-and-bound developed algorithm which are because of the the complexity of the studied model.

Key words

Combinatorial optimisation, Knapsack, Heuristics, Tabu search, Optimal methods.