Heuristiques hybrides pour la résolution de problèmes en variables 0-1 mixtes

Christophe Wilbaut

Université de Valenciennes et du Hainaut-Cambrésis

Numéro d'ordre : 06/12

Heuristiques Hybrides pour la Résolution de Problèmes en Variables 0-1 Mixtes

THÈSE

présentée et soutenue publiquement le 29 septembre 2006
pour l'obtention du

Doctorat de l'université de Valenciennes et du Hainaut-Cambrésis

Spécialité Automatique et Informatique des Systèmes Industriels et Humains

Discipline : Informatique

par

Christophe WILBAUT

Rapporteurs : Pr Mhand HIFI, Université de Picardie - Jules Vernes, LaRIA
Pr Jacques TEGHEM, Faculté Polytechnique de Mons, MathRO

Examinateurs : Pr Arnaud FRÉVILLE, Université de Valenciennes et du Hainaut-Cambrésis, LAMIH ROI
Pr Frédéric SEMET, Université de Valenciennes et du Hainaut-Cambrésis, LAMIH ROI
Pr El-Ghazali TALBI, Université des Sciences et Technologies de Lille, LIFL

Directeur : Pr Saïd HANAFI, Université de Valenciennes et du Hainaut-Cambrésis, LAMIH ROI

Laboratoire d'Automatique, de Mécanique et d'Informatique Industrielles et Humaines –- UMR 8530

Remerciements

Je tiens tout d'abord à remercier Monsieur Saïd Hanafi, Professeur à l'Université de Valenciennes, d'avoir accepté d'être mon directeur de thèse et d'avoir encadré mes travaux de recherche. Nos nombreuses réunions ont permis de faire progresser ce travail grâce à un échange constructif et cette collaboration fut de mon point de vue véritablement enrichissante.

Je remercie également Messieurs Mhand Hifi, Professeur à l'Université d'Amiens, et Jacques Teghem, Professeur à l'Université de Mons, qui m'ont fait l'honneur de rapporter ce travail.

Je remercie aussi Messieurs Arnaud Fréville, Professeur à l'Université de Valenciennes et Directeur de la recherche, de l'enseignement supérieur et des TIC au Conseil Régional du Nord-Pas de Calais, Frédéric Semet, Professeur à l'Université de Valenciennes, et El-Ghazali Talbi, Professeur à l'Université de Lille, d'avoir accepté de participer à mon jury.

Je profite de ces quelques lignes pour remercier l'ensemble des personnes que j'ai côtoyé durant cette période au sein du LAMIH dirigé par Monsieur Éric Markiewicz. J'insisterai plus particulièrement sur les membres de l'équipe Recherche Opérationnelle et Informatique dirigée par Frédéric Semet. Ne pouvant citer tout le monde, je me contenterai de nommer Madame Corinne Aureggi et Monsieur Philippe Dos Santos pour leur aide en ce qui concerne les aspects administratifs et les aspects matériels liés à ma thèse.

Merci aussi à l'ensemble des doctorants (ou ex-doctorants) de l'équipe avec lesquels j'ai été en contact durant cette période, et en particulier à ceux avec lesquels j'ai passé le plus de temps : Frédéric Beugnies, Célia Boulanger, Samuel Colin, Éric Duchenne, Nicolas Inglart, Fabien Malca et Mickaël Wiart. Merci à eux pour la bonne ambiance dans notre équipe, et merci à Éric de m'avoir supporté dans le bureau.

Je n’oublie pas de remercier ceux qui ont eu le courage de relire ce manuscrit et qui m’ont ainsi permis de l’améliorer : Alexandra, Éric, Marie, Saïd et Thomas.

Je ne pourrais terminer ces remerciements sans en adresser à ma famille et à ma future femme, Alexandra, à la fois pour la patience dont ils ont su faire preuve avec moi et pour leur soutien sans lequel ce travail n’aurait peut être pas eu le même aboutissement.

Résumé

Les problèmes d'optimisation en variables 0-1 mixtes permettent de modéliser de nombreux problèmes réels difficiles à résoudre. Cette thèse s'intéresse à la mise en œuvre de méthodes de résolution hybrides pour obtenir des solutions de bonne qualité en des temps raisonnables pour ces problèmes. L'ensemble des algorithmes présentés dans cette thèse est testé sur le problème du sac-à-dos multidimensionnel. Il consiste à maximiser une fonction linéaire en respectant un ensemble de contraintes linéaires.

Après une présentation de quelques concepts fondamentaux utilisés en recherche opérationnelle pour résoudre les problèmes d'optimisation, nous présentons dans le premier chapitre différents problèmes de la famille du sac-à-dos. Nous abordons dans le second chapitre un ensemble de méthodes efficaces existantes pour résoudre le problème du sac-à-dos multidimensionnel.

Nous proposons dans le chapitre 3 une première méthode hybride qui combine la programmation dynamique et la recherche tabou au sein d'un processus dit d'intensification globale. Des concepts de réduction sont également intégrés dans la programmation dynamique de manière à essayer de réduire la taille du problème.

La seconde approche décrite dans le chapitre 4 combine la recherche dispersée avec des éléments de la recherche tabou et des chemins reliants pour affiner la recherche. Une étude expérimentale est menée pour mesurer l'impact de différents composants de l'algorithme.

Nous terminons dans le chapitre 5 par une méthode utilisant conjointement la relaxation en continu et la relaxation en nombres entiers mixtes pour résoudre efficacement les problèmes en variables 0-1. Un ensemble de résultats numériques est présenté pour chacune de ces méthodes. La dernière approche permet d'améliorer quelques meilleures valeurs connues sur des instances existantes du problème du sac-à-dos multidimensionnel.

Mots clés : Problèmes en variables 0-1 mixtes, Sac-à-dos multidimensionnel, Métaheuristique, Méthode hybride, Recherche tabou, Recherche dispersée.

Laboratoire : LAMIH-UMR CNRS 8530, UVHC, "Le Mont Houy", 59313 Valenciennes Cedex 9, France

Abstract

The 0-1 mixed integer programs represent many difficult real problems. The subject of this thesis is the proposition of hybrid methods to obtain good solutions in reasonable time for these problems. The algorithms presented in this thesis are tested on the multidimensional knapsack problem. It consists on the maximisation of a linear function with the respect of a set of linear constraints.

In the first chapter of this thesis, we give a few concepts used to solve optimisation problems. We also present some problems of the knapsack problem family. We then present in the second chapter a state of the art to solve the multidimensional knapsack problem.

We propose in the third chapter a first hybrid method. It combines a dynamic programming approach with a tabu search algorithm in a global intensification process. Some reduction rules are also integrated in the dynamic programming phase to try to reduce the size of the problem.

The second approach is described in the chapter 4. It combines a scatter search algorithm with tabu search and path relinking components to enhance the process. We present an experimental study to assess the impact of some elements of the algorithm.

We finally present in the chapter 5 some heuristics using jointly the linear programming relaxation and the mixed integer linear programming relaxation to solve the 0-1 integer programs. A set of computational results is presented for each approach. The last one improves some best-known values on a set of available instances of multidimensional knapsack problem.

Title: Hybrid heuristics for 0-1 integer programming

Keywords: 0-1 Mixed Integer Programming, Multidimensional Knapsack Problem, Metaheuristic, Hybrid Method, Tabu Search, Scatter Search.

Tables des Matières

Table des Figures

Liste des Tableaux

Liste des Algorithmes

Liste des Notations

n	le nombre de variables du problème
$N = \{1,\ldots, n\}$	l'ensemble des variables du problème
m	le nombre de contraintes du problème
$M = \{1,\ldots,m\}$	l'ensemble des contraintes du problème
c_j	le coefficient de l'objet j dans l'objectif du problème
a_{ij}	le poids de l'objet j pour la contrainte i
A^j	le vecteur des poids de l'objet j pour le problème
b_i	le membre droit de la contrainte i
$v(P)$ ou v^*	la valeur optimale du problème P
$\bar{v}$ ou $\bar{v}(P)$	une borne supérieure du problème P
$\underline{v}$ ou $\underline{v}(P)$	une borne inférieure du problème P
$x^* = (x_1^*,\ldots, x_n^*)$	une solution optimale du problème
cx	la valeur de la solution x
$A(x)$	l'ensemble des poids associés à la solution x
$LP(P)$	la relaxation en continu du problème P

Introduction

Ce mémoire présente une synthèse de l'ensemble de mes travaux de recherche effectués lors de mon doctorat au sein du LAMIH[1] (équipe ROI[2]). Ces travaux s'inscrivent dans le cadre de l'optimisation en nombres entiers, et plus particulièrement dans celui des problèmes en variables binaires. Certaines approches proposées peuvent être assez facilement étendues à des problèmes d'optimisation en variables entières et/ou mixtes.

La résolution de problèmes d'optimisation intervient de nos jours dans de nombreux contextes, et les solutions obtenues par des algorithmes spécifiques pour résoudre ces problèmes doivent être fournies en des temps raisonnables en pratique. Cela est le cas en particulier si le nombre de problèmes à résoudre au sein d'une application a tendance à être important, ou si les solutions doivent être utilisées lors d'une interaction avec l'utilisateur (ou le preneur de décisions), comme dans des applications en temps réel. Un exemple classique de problème faisant appel à ce type de contraintes liées au temps de réponse est le problème de déploiement et redéploiement de flottes de véhicules. De nombreux problèmes réels peuvent être modélisés sous la forme d'un programme linéaire, et même si ces problèmes réels diffèrent en partie des problèmes « théoriques » rencontrés le plus souvent dans la littérature, il est généralement possible de se rapprocher d'un modèle existant pour lequel différentes approches de résolution efficaces ont été proposées. L'étude de ces modèles théoriques reste donc d'actualité puisqu'elle permet aux preneurs de décisions d'avoir un ensemble de méthodes efficaces qu'ils peuvent adapter pour résoudre leurs problèmes particuliers.

La définition d'une méthode de résolution exacte efficace, c'est-à-dire permettant d'obtenir une solution optimale du problème (une solution qu'aucune autre solution ne peut améliorer) rapidement avec une complexité spatiale raisonnable, représente pour certaines classes de problèmes d'optimisation un objectif difficile à atteindre. Cela est en particulier le cas lorsque les instances sont de grande taille. La complexité spatiale et/ou temporelle de ce type de méthodes rend leur application généralement difficile en pratique, voire impossible, et

[1] Laboratoire d'Automatique, de Mécanique et d'informatique Industrielles et Humaines, UMR CNRS 8530
[2] Recherche Opérationnelle et Informatique

cela malgré le développement incessant des outils informatiques. Devant ce constat, d'autres approches dites heuristiques (ou algorithmes approchés) sont apparues il y a plusieurs décennies. Ces algorithmes permettent dans la plupart des cas la génération d'une solution réalisable du problème rapidement mais le prix à payer est qu'ils n'assurent pas toujours un niveau de qualité minimum. Certaines heuristiques sont cependant construites de manière à pouvoir assurer une distance maximum (le terme de saut est souvent rencontré dans la littérature) par rapport à une solution optimale du problème.

Devant les limites rencontrées par les heuristiques pour obtenir une solution réalisable de bonne qualité pour certains types de problèmes, d'autres approches appelées *métaheuristiques* sont apparues. Ces algorithmes sont plus complets et complexes qu'une simple heuristique, et permettent généralement d'obtenir une solution de très bonne qualité pour le problème traité. Cela entraîne nécessairement une augmentation des complexités spatiale et/ou temporelle par rapport aux heuristiques. Le rapport entre la qualité de la solution finale et le temps d'exécution d'une métaheuristique reste cependant dans la majorité des cas très intéressant par rapport aux autres types d'approches de résolution.

Finalement une dernière classe d'algorithmes peut être trouvée dans la littérature pour résoudre des problèmes d'optimisation. Le principe consiste à combiner des algorithmes exacts et/ou des algorithmes approchés pour essayer de tirer profit des points forts de chaque approche et améliorer le comportement global de l'algorithme. La terminologie rencontrée dans la littérature est l'hybridation de méthodes. L'objectif du travail présenté dans ce mémoire est de proposer plusieurs approches efficaces combinant des méthodes exactes et/ou des heuristiques ou des métaheuristiques pour obtenir des solutions de bonne qualité en des temps d'exécution raisonnables pour les problèmes en variables binaires.

De manière à pouvoir juger l'efficacité de nos propositions, nous avons testé toutes nos approches sur le même problème, à savoir le problème du sac-à-dos multidimensionnel en variables 0-1. Le choix de ce problème s'explique par les raisons suivantes. Le problème du sac-à-dos et toutes ses variantes sont des problèmes de la classe NP-difficile. L'application de méthodes exactes sur ces problèmes ne permet généralement pas d'obtenir une solution optimale en des temps raisonnables lorsque leur taille augmente. Ils permettent également de formuler de nombreux problèmes réels et sont aussi utilisés comme sous-problèmes d'autres problèmes. Le problème du sac-à-dos multidimensionnel est un cas particulier de la programmation en nombres entiers. Il représente à la fois un ensemble de problèmes et reste un problème particulier difficile à résoudre. Finalement, l'existence de nombreux jeux de données disponibles sur Internet permet de pouvoir comparer les résultats obtenus avec ceux

d'autres approches existantes dans la littérature, et le fait que des problèmes de taille moyenne (avec 500 variables et 10 contraintes) ne puissent toujours pas être résolus en des temps raisonnables de manière optimale par des logiciels performants nous a poussé à essayer d'améliorer les résultats obtenus pour ce problème. Le **chapitre 1** de ce mémoire traite ainsi de la famille des problèmes de type sac-à-dos. Ce chapitre nous permet de situer le contexte de ce travail en apportant quelques éléments de base utilisés en recherche opérationnelle nécessaires à la compréhension de la suite de ce mémoire. Nous donnons également quelques références existantes pour résoudre ces problèmes.

Nous abordons dans le **chapitre 2** plus spécifiquement le problème du sac-à-dos multidimensionnel et un ensemble d'approches de résolution efficaces proposées pour le résoudre. Nous mettons en évidence les algorithmes les plus connus et réputés pour ce problème.

Nous décrivons dans le **chapitre 3** un algorithme hybride utilisant la programmation dynamique et la recherche tabou. Une approche basée sur la programmation dynamique avec l'utilisation d'une liste est utilisée comme une heuristique (dans le sens où elle n'est pas appliquée à tout le problème) pour fournir des solutions optimales de sous-problèmes du problème initial. L'algorithme complet est un processus dit d'intensification globale au cours duquel la recherche tabou utilise les informations générées par la programmation dynamique lors de l'évaluation du voisinage de la solution courante. Différents concepts de réduction sont également intégrés dans l'algorithme de programmation dynamique pour réduire la taille du problème initial.

Dans le **chapitre 4** nous décrivons un algorithme évolutif de type recherche dispersée. Après une présentation de la méthode et de quelques références existantes, nous abordons les différents composants de l'algorithme que nous mettons en œuvre. Nous présentons ensuite une étude expérimentale réalisée sur un petit ensemble d'instances de problèmes de sac-à-dos multidimensionnel. Cette étude nous permet de mettre en évidence l'impact de certains composants sur les performances d'un algorithme de recherche dispersée. Nous proposons également un générateur de solutions élites qui produit une population initiale de bonne qualité pour ce problème. L'influence de cette population initiale est montrée grâce aux résultats numériques finaux qui sont du même ordre qu'un algorithme génétique performant pour le problème du sac-à-dos multidimensionnel.

Nous présentons finalement dans le **chapitre 5** différentes heuristiques hybrides basées sur la relaxation en continu et sur la relaxation en nombres entiers mixtes pour la résolution de problèmes en variables 0-1. Ce travail se base sur une approche initialement

proposée à la fin des années 1970, et différentes améliorations et extensions sont proposées. Nous définissons aussi deux nouvelles heuristiques nous permettant d'améliorer certaines des meilleures solutions actuelles sur des instances existantes difficiles et corrélées du problème du sac-à-dos multidimensionnel. Les heuristiques obtenues peuvent également être utilisées comme des algorithmes exacts, même si dans ce cas la complexité temporelle devient rapidement très importante.

Nous terminons ce mémoire en présentant nos conclusions générales et les perspectives de recherche.

Chapitre 1

Problèmes de la famille du sac-à-dos

Nous présentons dans ce chapitre le contexte de nos travaux, à savoir la résolution de problèmes d'optimisation en variables 0-1 mixtes. La première partie du chapitre est consacrée à la description de quelques éléments élémentaires utilisés en recherche opérationnelle. Nous abordons ensuite un ensemble de problèmes de la famille du problème du sac-à-dos. Leur description nous permet de mettre en avant le nombre important d'applications existantes. La plupart de ces problèmes ont été et sont encore très étudiés par la communauté des chercheurs en recherche opérationnelle. Ils sont souvent facilement compréhensibles grâce à une formulation simple. Nous nous contentons de présenter quelques applications et quelques approches existantes pour les résoudre, et insistons aussi sur certains d'entre eux encore peu étudiés ou récemment proposés.

Nous commençons notre étude bibliographique par le problème du sac-à-dos multidimensionnel et ses cas particuliers. Les méthodes de résolution du problème du sac-à-dos multidimensionnel sont abordées dans le chapitre suivant de ce mémoire. Nous décrivons ensuite d'autres problèmes obtenus en ajoutant ou en modifiant les contraintes du problème du sac-à-dos multidimensionnel. Nous concluons ce chapitre par des problèmes obtenus en modifiant l'objectif du problème du sac-à-dos multidimensionnel.

1.1 Introduction à la programmation en nombres entiers mixtes

La prise de décisions dans la vie de tous les jours dépend souvent de nombreux paramètres basés par exemple sur l'émotion, la personne qui prend la décision, son état physique et/ou moral au moment de la prise de décision, *etc*. La prise en compte de ces paramètres est très difficile, et leur impact peut être important. Un problème de décision peut généralement être représenté par un problème d'optimisation dans lequel il faut affecter des valeurs numériques à des variables. Le but du problème, appelé l'objectif, peut par exemple

consister à obtenir un profit maximum, ou alors à minimiser les pertes si nous nous plaçons dans un contexte financier. Il est alors modélisé sous la forme d'un problème de maximisation ou de minimisation.

Le fait de modéliser ces problèmes et de les résoudre par un programme informatique permet entre autres d'automatiser le processus de prise de décision et de simplifier la tâche du preneur de décisions. Cependant, le fait de trouver une configuration meilleure que toutes les autres (cette configuration est appelée configuration optimale), peut rapidement devenir très difficile car l'ensemble des options possibles tend à être très important dans la réalité.

Le but de la recherche opérationnelle est de permettre la résolution de problèmes concrets représentés par des modèles mathématiques. Cette résolution se fait dans la plupart des cas par un ordinateur, une résolution *humaine* étant généralement impossible de par la complexité des problèmes traités.

1.1.1 Programmation linéaire en nombres entiers mixtes

La forme la plus simple d'une prise de décision est le choix entre deux alternatives. Une telle décision, dite binaire, est modélisée en introduisant une variable binaire qui peut prendre les valeurs 0 ou 1. La résolution d'un problème de décision consiste alors à choisir une des deux alternatives pour l'ensemble des variables du problème. Celles-ci sont les variables de décision, et l'affectation d'une valeur à une variable peut avoir une influence sur les autres variables.

Une solution est représentée par un ensemble de variables de décision. Elle peut être évaluée à l'aide d'une combinaison de valeurs associées à chacune des variables dans le cas le plus simple. Cette évaluation des solutions est faite grâce à la fonction objectif du problème. Cette fonction peut être linéaire ou non linéaire (quadratique par exemple). Un problème est aussi généralement composé de contraintes sur les variables, ce qui permet de réduire l'espace des solutions réalisables du problème. Ces contraintes définissent un ensemble de règles à respecter par une solution dite réalisable du problème. Le but final du problème est de trouver une solution, la *meilleure* possible au sens de la fonction objectif, et qui respecte l'ensemble des contraintes.

Pour illustrer ces notions, prenons un exemple dans lequel la prise de décision consiste à choisir entre l'utilisation d'une voiture ou d'un train pour faire un déplacement. Nous pouvons représenter ce choix par une variable qui vaut 1 si le mode de transport choisi est la voiture, et 0 si le mode choisi est le train. Les valeurs associées aux variables dans l'objectif

peuvent par exemple correspondre aux préférences. Nous pouvons ainsi imaginer que la préférence pour la voiture est inférieure à celle du train selon un critère de confort. Nous pouvons aussi supposer que le fait de prendre le train impose des contraintes d'horaires plus fortes que le fait de prendre la voiture. Ce genre de contraintes peut être modélisé sur des horaires de parcours par exemple.

Les concepts de base sur l'optimisation combinatoire et la recherche opérationnelle présentés dans cette section peuvent être trouvés sous différentes formes dans les ouvrages de Nemhauser et Wolsey [NW99] ou de Kellerer, Pferschy et Pisinger [KPP04] par exemple.

Lorsque l'objectif d'un problème et l'ensemble de ses contraintes peuvent être entièrement modélisés à l'aide de fonctions, ce problème peut être représenté par un programme linéaire (*PL*) sous la forme générale suivante

$$(PL) \quad \begin{cases} Opt & \sum_{j \in N} c_j x_j \\ \text{sous les contraintes :} & \sum_{j \in N} a_{ij} x_j \Delta_i b_i \quad \forall i \in M \\ & l_j \le x_j \le u_j \quad \forall j \in N \end{cases}$$

avec :

- *Opt* qui est soit égal à Minimiser (min) ou Maximiser (max) selon le problème traité.
- $N = \{1,\ldots,n\}$ l'ensemble des variables de décision.
- $M = \{1,\ldots,m\}$ l'ensemble des contraintes du problème.
- x un vecteur associé à une solution du problème : $x = (x_1,x_2,\ldots,x_n)$.
- $\Delta_i \in \{\le; =; \ge\} \ \forall i \in M$.
- l_j (respectivement u_j) la borne inférieure (resp. supérieure) sur la valeur de la variable x_j.

Dans toute la suite de ce mémoire nous nous plaçons dans le cadre de l'optimisation de problèmes de maximisation (sauf mention contraire). Ces problèmes sont représentés de manière générale avec $\Delta_i = \le, \ \forall i \in M$.

Nous définissons l'ensemble des coefficients dans l'objectif du problème par le vecteur $c = (c_1,c_2,\ldots,c_n)$, et l'ensemble des coefficients associés aux m contraintes du problèmes par la matrice $A = (a_{ij})_{i \in M, j \in N}$.

Dans le programme *PL*, il n'y a pas de contraintes d'intégrité. Il définit un programme linéaire en variables continues. Lorsqu'une partie des variables sont contraintes à être entières, le problème obtenu est dit en variables mixtes.

D'autres types de problèmes sont définis à partir du programme *PL*. Lorsque toutes les variables sont entières, le programme obtenu est appelé programme en variables entières (*PE*).

$$(PE) \quad \begin{cases} \max & \sum_{j \in N} c_j x_j & \\ s.c. & \sum_{j \in N} a_{ij} x_j \le b_i & \forall i \in M \\ & l_j \le x_j \le u_j & \forall j \in N \\ & x_j \in \mathbb{Z} & \forall j \in N \end{cases}$$

Finalement, lorsque toutes les variables sont de type booléen, le programme obtenu (0-1*PE*) est un cas particulier de (*PE*).

$$(0-1PE) \quad \begin{cases} \max & \sum_{j \in N} c_j x_j & \\ s.c. & \sum_{j \in N} a_{ij} x_j \le b_i & \forall i \in M \\ & x_j \in \{0,1\} & \forall j \in N \end{cases}$$

Le travail présenté dans ce mémoire s'intéresse en particulier à la résolution des problèmes représentés par (0-1*PE*) pour lesquels toutes les données sont supposées entières. Nous utiliserons également la représentation équivalente suivante :

$$(0-1PE) \quad \begin{cases} \max & cx \\ s.c. & Ax \le b \\ & x \in \{0,1\}^n \end{cases} \tag{1.1}$$

Dans l'ensemble de ce mémoire nous utilisons les notations répertoriées dans la liste des notations. Certaines notations supplémentaires, propres à un chapitre particulier, sont spécifiées au moment opportun.

1.1.2 Complexité et méthodes de résolution

Pour comparer les performances de plusieurs algorithmes et dire si un algorithme est meilleur qu'un autre pour résoudre un problème donné, nous pouvons évaluer leur complexité dans le pire des cas.

La complexité consiste à estimer le temps de calcul d'un algorithme (complexité temporelle) et les besoins en mémoire informatique (complexité spatiale). La théorie de la complexité repose sur différentes classes de complexité permettant de créer un ordre sur les algorithmes proposés pour résoudre un problème. Parmi les classes de complexité existantes, citons la classe *P*. Un problème de décision est dans la classe *P* s'il peut être décidé par un algorithme déterministe en un temps *polynomial* par rapport à la taille de l'instance. On qualifie alors le problème de polynomial.

La plupart des problèmes d'optimisation combinatoire appartiennent à la classe des problèmes NP-difficile. Cela signifie qu'il est peu probable qu'il existe un algorithme permettant de le résoudre en temps polynomial (à moins que les classes *P* et *NP* ne soient confondues). La théorie de la complexité s'intéresse ainsi de manière générale à l'étude de la difficulté des problèmes (voir Garey et Johnson [GJ79]).

Pour résoudre un problème d'optimisation, il existe plusieurs types d'approche. Une méthode exacte est un algorithme qui permet l'obtention d'au moins une solution optimale du problème à résoudre. Une fois une solution optimale obtenue, l'algorithme doit également être capable d'en prouver l'optimalité, ce qui peut parfois être tout aussi difficile à faire que d'obtenir cette solution. Ce type d'algorithme nécessite généralement un temps d'exécution important et/ou d'importantes ressources mémoires sur des instances de grande taille. La complexité de ces algorithmes est donc souvent trop importante pour qu'ils puissent être appliqués sur des problèmes comprenant de nombreuses variables et/ou contraintes. Sans rentrer dans le détail, nous pouvons citer quelques approches exactes classiques utilisées en recherche opérationnelle sur lesquelles nous revenons dans la suite de ce mémoire :

- La programmation dynamique ;
- Les méthodes de séparation et évaluations ;
- Les approches polyhédrales.

Lorsque l'algorithme utilisé ne permet pas nécessairement d'obtenir une solution optimale du problème, la méthode est appelée méthode approchée ou heuristique. Ce genre d'approche est souvent adapté au problème à résoudre, et elle permet généralement la génération d'une solution réalisable de bonne qualité en des temps de calcul peu importants et en nécessitant également une charge mémoire limitée. L'expression « de bonne qualité » signifie ici que l'algorithme permet de trouver une solution de valeur assez proche de la valeur optimale du problème. Les heuristiques sont très utiles pour traiter les problèmes de

grande taille, ou encore pour générer des solutions rapidement dans un contexte spécifique tel que les applications en temps réel.

Devant la limite des résultats des heuristiques pour certains problèmes, d'autres approches communément appelées *métaheuristiques* sont apparues. Une métaheuristique peut être vue comme un processus incluant souvent plusieurs heuristiques. Elle les guide vers une solution proche d'une solution optimale du problème à l'aide de différentes stratégies pour utiliser l'information générée au cours de la recherche. Elle permet aussi d'éviter que le processus ne reste bloqué dans ce qui est appelé un optimum local. Une des particularités des métaheuristiques est qu'elles sont souvent générales et adaptables à de nombreux problèmes, alors qu'une heuristique est plus spécifique à un problème donné. Plusieurs définitions ont été proposées par Osman et Laporte [OL96] ou encore Pirlot [Pir02]. Citons ici simplement quelques métaheuristiques connues que nous abordons dans la suite :

- La recherche tabou ;
- Le recuit simulé ;
- La recherche dispersée ;
- Les algorithmes génétiques ; *etc*.

1.2 Le sac-à-dos multidimensionnel et ses cas particuliers

Le problème du sac-à-dos est très connu en recherche opérationnelle depuis de nombreuses années, et il est encore très étudié de nos jours. Sa formulation simple et sa complexité de résolution en ont fait un problème majeur pour introduire le domaine de l'optimisation combinatoire et les méthodes de résolutions existantes pour les problèmes d'optimisation.

1.2.1 Le problème du sac-à-dos en variables 0-1

Le problème du sac-à-dos (*SAD*) peut être modélisé sous la forme d'un programme linéaire comportant une fonction linéaire et une contrainte linéaire de la manière suivante

$$(SAD) \qquad \left[\begin{array}{ll} & \max \sum_{j \in N} c_j x_j \\ s.c. & \sum_{j \in N} a_j x_j \leq b \\ & x_j \in \{0,1\} \qquad \forall j \in N = \{1,...,n\} \end{array} \right.$$

avec c_j, a_j $\forall j \in N$, et b entiers positifs.

Ce programme linéaire est défini sur $n = |N|$ variables. Le profit associé à la variable x_j pour j dans N est donné par le coefficient c_j, et chaque objet a un poids représenté par le coefficient a_j. La somme des poids des variables de décision fixées à 1 ne peut excéder la capacité totale du sac-à-dos, b.

Sans perdre en généralités, nous pouvons supposer que la contrainte (1.2) est également respectée.

$$\max\{a_j : j \in N\} \leq b < \sum_{j \in N} a_j \tag{1.2}$$

Le membre droit de la contrainte (1.2) signifie que la somme des poids associés aux n variables de décision doit être supérieure à la capacité b du sac car une solution optimale consisterait simplement à fixer toutes les variables à 1 dans le cas contraire. Le membre gauche signifie que le poids de chaque objet doit être inférieur à la capacité totale du sac de manière à ne pas pouvoir fixer immédiatement la variable correspondante à 0.

L'appellation *sac-à-dos* est une interprétation du programme linéaire *SAD*. Considérons un vacancier qui a à sa disposition un sac-à-dos. Il établit une liste d'objets importants qu'il est susceptible de prendre et qui lui seraient utiles. Chacun de ces objets, numérotés de 1 à n, lui apporte un certain confort, noté c_j, et chaque objet a un certain poids ou prend une certaine place dans le sac, a_j. La capacité du sac n'étant pas infinie (elle est définie par b), le problème consiste pour le vacancier à maximiser son confort en respectant la contrainte de capacité.

Le problème *SAD* a de nombreuses applications pratiques. Un des premiers exemples de modélisation d'un problème sous la forme *SAD* remonte à 1897 par Mathews [Mat97]. L'auteur montre comment agréger plusieurs contraintes en une seule. Le problème *SAD* permet entre autres de modéliser des problèmes de chargement de cargos, de gestion de stocks, de sélection de projets et de budgets. La sélection de projets ou d'investissements représente le plus grand nombre d'applications pratiques de ce problème. Les articles de Lorie et Savage [LS55] et de Nemhauser et Ullmann [NU69] en sont des exemples. Le problème *SAD* a aussi été utilisé pour générer automatiquement des inégalités couvrantes par Balas [Bal75] et Wolsey [Wol75] par exemple. Mentionnons pour terminer l'utilisation du problème *SAD* dans les systèmes de cryptographie par Merkle et Hellman [MH78].

Nous abordons maintenant quelques méthodes proposées pour résoudre le problème *SAD*. Certaines d'entre elles ont été étendues au problème du sac-à-dos multidimensionnel.

Le problème *SAD* est le problème de décision en nombres entiers non trivial le plus simple par sa formulation. Il ne comporte qu'une seule contrainte (hormis les contraintes d'intégrité, i.e. $x_j \in \{0,1\}$, $\forall j \in N$), et des coefficients positifs. Il représente toutefois un challenge de par sa complexité puisqu'il appartient à la classe NP-difficile.

En effet, il n'existe pas d'algorithme de complexité polynomial pour résoudre le problème *SAD*. Cependant, d'autres types d'algorithmes, appelés schémas d'approximation pseudo polynomiaux ont été proposés pour ce problème. Introduisons tout d'abord quelques concepts pour pouvoir définir un schéma d'approximation polynomial.

Soit *A* un algorithme pour résoudre un problème d'optimisation. Cet algorithme est un algorithme approximatif avec une performance garantie égale à ε ($\varepsilon \in]0,1[$) si et seulement si, pour chaque instance *I* du problème, l'inégalité (1.3) suivante est respectée

$$\frac{v(I) - cx}{v(I)} \leq \varepsilon \tag{1.3}$$

avec *x* la solution fournie par *A* pour l'instance *I*, *v*(*I*) la valeur optimale de l'instance *I* et *cx* la valeur de l'objectif au point *x*.

Un algorithme *A* est un schéma d'approximation ε pour un problème donné si l'inégalité (1.4) est vérifiée pour chaque valeur de ε dans]0,1[, chaque instance *I* du problème, et avec *x* la solution obtenue par *A* pour *I*.

$$(1 - \varepsilon)v(I) \leq cx \tag{1.4}$$

Un schéma d'approximation ε est un schéma d'approximation polynomiale si son temps d'exécution est polynomial en *n*. Lorsqu'un tel algorithme existe pour un problème, cela permet de dire que ce problème n'est pas fortement NP-difficile.

Lorsqu'un schéma d'approximation ε a un temps d'exécution polynomial en *n* et en $1/\varepsilon$, il est dit totalement polynomial pour le problème. Ce type d'algorithme est ce qui peut être obtenu de mieux pour un problème NP-difficile (à moins que *P* = *NP*).

Différents schémas d'approximation polynomiaux ont été proposés pour le problème *SAD*. Dans l'algorithme H^{ε} présenté par Kellerer, Pferschy et Pisinger [KPP04], les objets sont tout d'abord ajoutés par sous-ensembles dans le sac. S'il reste de la place dans le sac, le but est ensuite d'essayer de compléter la solution avec d'autres objets à l'aide d'une heuristique de type glouton (ce genre d'heuristique est présenté plus loin dans ce mémoire).

Des schémas d'approximation totalement polynomiaux ont également été proposés pour le problème *SAD*. Ces algorithmes ne sont pas toujours utilisables en pratique car ils

requièrent un espace mémoire très important. L'algorithme proposé par Kellerer et Pferschy [KP99] est un exemple de schéma d'approximation totalement polynomial utilisable en pratique. D'autres algorithmes peuvent être trouvés comme celui de Ibarra et Kim [IK75] par exemple.

De nombreux algorithmes exacts sont recensés dans la littérature pour le problème *SAD*. Nous commençons par présenter rapidement le principe des algorithmes de séparation et évaluations avant de donner quelques exemples d'approches basées sur cette méthode pour ce problème. Les algorithmes de séparation et évaluations consistent à effectuer une énumération implicite représentée par une méthode arborescente en séparant le problème en sous-problèmes (deux ou plus). Chaque sous-problème est évalué (à l'aide d'une relaxation) de manière à supprimer de l'espace de recherche les sous-problèmes ne pouvant contenir de solution optimale du problème. Cette recherche peut être représentée graphiquement par un arbre. Les méthodes de séparation et évaluations reposent sur trois grands principes énoncés par Mitten [Mit70] :

- L'évaluation de chaque sous-problème permet d'appliquer un ensemble de tests pour déterminer si l'exploration du sous-ensemble de solutions associé à ce sous-problème doit être continuée ou pas.
- La séparation permet de décomposer un problème donné en au moins deux sous-ensembles de cardinalité inférieure.
- La stratégie de parcours permet de choisir quel est le sommet à séparer dans la liste des sommets générés par l'exploration.

Ce genre d'approche consiste à énumérer l'ensemble de toutes les solutions possibles du problème. La complexité d'un tel algorithme est en $O(2^n)$ pour un problème en variables binaires. Pour l'améliorer, différents moyens peuvent être mis en place de manière à effectuer une énumération dite « intelligente » de l'espace des solutions en supprimant certaines branches de l'arbre de recherche. En effet généralement seul un petit nombre de solutions réalisables sont énumérées explicitement. Il faut cependant bien entendu veiller à ce que les parties de l'espace de recherche non explorées ne puissent contenir de solutions optimales.

Différents algorithmes de séparation et évaluations ont été proposés pour le problème *SAD*. Le calcul des bornes inférieures et supérieures associées à chaque sous-espace est un point important permettant d'affiner les performances de ces algorithmes. Parmi les

algorithmes existants, nous citerons par exemple ceux de Martello et Toth [MT77, MT90] où les auteurs proposent d'utiliser l'énumération partielle pour générer des bornes supérieures de bonne qualité. Fayard et Plateau [FP82] ont également proposé le même type de bornes supérieures. Pour déterminer les bornes inférieures, Pisinger [Pis95a] a proposé différentes heuristiques basées sur la résolution du problème *noyau* (*core problem* selon la terminologie anglaise dont nous reparlons plus loin dans ce mémoire). Ces heuristiques peuvent être vues comme des améliorations des heuristiques de type glouton. Enfin, un des algorithmes les plus connus est dû à Horowitz et Sahni [HS74]. Les auteurs proposent une version récursive d'un algorithme de séparation et évaluations dans lequel chaque itération correspond à un branchement sur la variable libre la plus « efficace », au sens des algorithmes gloutons.

D'autres algorithmes exacts qui reposent sur la programmation dynamique due à Bellman [Bel57] peuvent aussi être recensés. Nous reviendrons dans le chapitre 3 sur une présentation de la programmation dynamique puisque nous avons utilisé cette approche dans une méthode de résolution hybride appliquée au problème du sac-à-dos multidimensionnel.

Les algorithmes proposés pour le problème *SAD* par Ahrens et Finke [AF75] et par Toth [Tot80] utilisent les points de discontinuité associés à la valeur de la fonction sac-à-dos définie à partir de la relation de récurrence pour ce problème. Elle est en effet croissante par morceaux et reste constante entre deux points de discontinuité. Il est donc possible de ne conserver que ces points sans perdre d'information. Ils intègrent également des techniques de dominance entre paire d'éléments. Horowitz et Sahni [HS74] ont proposé une approche basée sur la décomposition du problème de départ en deux sous-problèmes. Pour chacun des deux sous-problèmes est établi une liste de paires non dominées, et finalement les deux listes sont fusionnées pour trouver une solution optimale du problème initial. La programmation dynamique a aussi souvent été hybridée avec une autre approche de résolution. L'algorithme de Marsten et Morin [MM78], proposé pour la résolution de problèmes mathématiques et qui combine la méthode de séparation et évaluations et la programmation dynamique, a été appliqué au problème *SAD* par Toth [Tot80] et a permis l'obtention de bons résultats par exemple.

1.2.2 Problème de la somme d'un sous-ensemble

Lorsque les coefficients associés à une variable sont identiques dans l'objectif du problème et dans la contrainte de capacité, le problème obtenu est appelé problème de la

somme d'un sous-ensemble (*SSS*) (*subset sum problem* en anglais). La formulation de ce problème est facilement obtenue à partir de celle du problème *SAD*.

$$(SSS) \quad \begin{cases} \max & \sum_{j \in N} a_j x_j \\ s.c. & \sum_{j \in N} a_j x_j \leq b \\ & x \in \{0,1\}^n \end{cases}$$

Le problème *SSS* est un cas particulier du problème *SAD*. Pisinger [Pis99a] l'utilise comme un sous-problème pour résoudre le problème du sac-à-dos multiple que nous abordons dans la section 1.3.1. Les solutions du problème *SSS* peuvent aussi être utilisées pour obtenir des bornes inférieures de bonne qualité pour des problèmes de planification (Guéret et Prins [GP99]).

Différents algorithmes proposés pour le problème *SAD* ont été appliqués ou adaptés pour le problème *SSS* et ont permis d'obtenir de bons résultats. L'algorithme hybride *MTS* utilisant à la fois la programmation dynamique et la méthode de séparation et évaluations de Martello et Toth [MT84] en est un exemple.

1.2.3 Problème du sac-à-dos bi-dimensionnel

Le problème du sac-à-dos bi-dimensionnel (*SADB*) est une extension du problème *SAD*. Il correspond au cas $m = 2$. Le problème *SADB* a souvent été traité comme un problème à part entière. Il ne comporte que deux contraintes, mais il est déjà très difficile à résoudre car il n'existe pas de schéma d'approximation complètement polynomial pour ce problème (voir Gens et Levner [GL79]).

Plusieurs travaux ont été menés en particulier sur le calcul de bornes supérieures de ce problème car les méthodes de relaxation sont généralement performantes. Les problèmes réduits se ramènent en effet à des instances de problèmes *SAD* ou des relaxations de problèmes *SAD*. Ainsi Thiongane, Nagih et Plateau [TNP03] ont par exemple proposé un algorithme basé sur le calcul d'un sous-gradient pour déterminer la valeur du dual Lagrangien du problème *SADB*.

Fréville et Plateau [FP93b, FP96] ont proposé plusieurs techniques pour résoudre de manière exacte le problème *SADB*. Leurs approches reposent en partie sur la résolution d'un problème dual agrégé (surrogate selon la terminologie anglaise. Nous utilisons ce terme dans la suite). La méthode est divisée en deux phases principales. Dans une première phase l'algorithme essaie de réduire le problème en fixant le plus de variables possible à l'aide de

bornes supérieures dérivées à partir de solutions du dual surrogate. Dans une seconde phase un algorithme de séparation et évaluations est exécuté avec les bornes supérieures générées précédemment. Martello et Toth [MT03] ont récemment proposé un algorithme exact permettant de résoudre de manière optimale une grande partie d'un ensemble d'instances de grande taille générées aléatoirement.

Parmi les autres références existantes, Hill et Reilly [HR00] ont par exemple étudié l'influence de la corrélation entre les coefficients de la fonction objectif et les deux valeurs des capacités du problème. Les résultats obtenus dépendent en partie des algorithmes utilisés et montrent qu'il est difficile de classifier une instance particulière d'un problème comme étant facile ou difficile.

1.2.4 Problème du sac-à-dos multidimensionnel en variables 0-1

Le problème du sac-à-dos multidimensionnel en variables 0-1 (*SADM*) est une généralisation du problème *SAD*. Il correspond au cas $m > 1$. Le problème *SADB* est donc un cas particulier du problème *SADM*. Le problème *SADM* peut être modélisé comme un programme linéaire de la manière suivante

$$(SADM) \qquad \left[\begin{array}{ll} \max \sum_{j\in N} c_j x_j & \\ s.c. \quad \sum_{j\in N} a_{ij} x_j \le b_i & \forall i \in M = \{1,...,m\} \\ x_j \in \{0,1\} & \forall j \in N = \{1,...,n\} \end{array}\right. \tag{1.5}$$

avec $c_j \ \forall j \in N$, $a_{ij} \ \forall i \in M$ et $\forall j \in N$ et $b_i \ \forall i \in M$ entiers positifs.

Sans perdre en généralités nous pouvons supposer par extension de (1.2) que la contrainte suivante est respectée.

$$\max\{a_{ij} : j \in N\} \le b_i < \sum_{j\in N} a_{ij} \qquad \forall i \in M \tag{1.6}$$

Notons qu'il existe différentes appellations de ce problème autre que sac-à-dos multidimensionnel comme problème du sac-à-dos multiple ou du sac-à-dos multi-contraints. Cependant, la majorité des auteurs utilisent l'expression sac-à-dos multidimensionnel, c'est pourquoi nous utiliserons cette appellation tout au long de ce mémoire.

Le problème *SADM* est un cas particulier de la programmation en nombres entiers avec une seule restriction sur les variables qui doivent être binaires et une sur les coefficients qui doivent être positifs. La particularité de ce problème par rapport aux problèmes en nombres entiers se justifie par deux aspects théoriques. Premièrement, la matrice des

contraintes associée au sac-à-dos multidimensionnel est généralement dense, à la différence de nombreuses classes de problèmes en nombres entiers. Deuxièmement, une solution réalisable du problème du sac-à-dos multidimensionnel est facilement obtenue en fixant toutes les variables à 0 (même si cette solution ne présente pas d'intérêt pratique), alors que l'obtention d'une solution réalisable peut être difficile pour certains problèmes en nombres entiers.

Le problème *SADM* est devenu depuis quelques années un problème très souvent utilisé pour tester de nouvelles métaheuristiques comme la recherche tabou (Dammayer et Voss [DV93], Glover et Kochenberger [GK96], Hanafi et Fréville [HF98], *etc*), et les algorithmes génétiques (Chu et Beasley [CB98]) en particulier. Ce problème est aussi facilement utilisable pour présenter la recherche opérationnelle en enseignement.

Le problème *SADM* fait partie de la classe des problèmes NP-Difficiles par extension du problème *SAD*. La difficulté augmente très sensiblement avec le nombre de contraintes. Ainsi, des instances comportant 500 variables et 10 contraintes ne peuvent être résolues de manière optimale en des temps raisonnables de nos jours par des logiciels d'optimisation efficaces. D'un point de vue pratique, la majorité des instances des problèmes *SADM* comportent un petit nombre de contraintes alors que le nombre de variables peut devenir important.

De nombreuses applications du problème *SADM* peuvent être trouvées dans la littérature. Ce problème est à la base un modèle de problèmes d'allocation de ressources. Lorie et Savage [LS55] et Manne et Markowitz [MM57] ont été les premiers à utiliser le problème *SADM* dans un contexte de modélisation de problèmes de budgets. Meier, Christofides et Salkin [MCS01] ont aussi utilisé ce problème de manière plus réaliste en combinant les modèles d'allocation de budgets avec de nouvelles méthodes d'évaluation de projets. Dans cette approche, le problème *SADM* est utilisé comme un sous-problème couplé avec des contraintes généralisées sur les bornes supérieures.

D'autres applications existent dans les domaines de la gestion de stocks (Gilmore et Gomory [GG66]) et de chargements (Bellman [Bel57], Shih [Shi79]). Le problème *SADM* a aussi été utilisé récemment pour modéliser un problème de planification de prises de photos par le satellite *SPOT* (Vasquez et Hao [VH01a]). Dans un contexte informatique, l'allocation de ressources dans les systèmes informatiques distribués (Gavish et Pirkul [GP82]), et la planification de programmes informatiques (Thesen [The73]) ont également été abordées.

Le problème *SADM* est aussi utilisé comme un sous-problème de plusieurs problèmes d'optimisation en nombres entiers. Ainsi par exemple, Gabrel, Knippel et Minoux [GKM99]

l'utilisent avec des contraintes de type choix multiples (nous décrivons cette variante du problème *SADM* dans la section 1.3.3) comme problème à résoudre à chaque itération d'un problème plus général d'optimisation de réseaux de marchandises. Il est aussi clairement un sous-problème du problème du sac-à-dos multidimensionnel avec contraintes de demande (voir section 1.3.4).

Les articles de Fréville [Fré04] et de Fréville et Hanafi [FH05] présentent de nombreuses références sur ce problème.

1.3 Le sac-à-dos multidimensionnel avec contraintes additionnelles

D'autres problèmes peuvent être obtenus à partir du problème *SADM* ou du problème *SAD* en changeant le type des contraintes, ou en ajoutant de nouvelles contraintes.

1.3.1 Sac-à-dos multiple

Le problème du sac-à-dos multiple (*SAD-M*) est une généralisation du problème *SAD*. Il correspond au cas d'un problème comprenant plusieurs sacs. Il est cependant différent du problème *SADM*. Il s'agit d'assigner chaque objet à un sac au plus en respectant les contraintes de capacité de chaque sac et en maximisant le profit total. Le problème *SAD-M* peut être modélisé par le programme linéaire suivant.

$$(SAD\text{-}M)\quad \left[\begin{array}{lll} \max & \sum_{i=1}^{m}\sum_{j=1}^{n} c_j x_{ij} & \\ s.c. & \sum_{j\in N} a_j x_{ij} \le b_i & \forall i = 1,...,m \\ & \sum_{i=1}^{m} x_{ij} \le 1 & \forall j = 1,...,n \\ & x_{ij} \in \{0,1\} & \forall i = 1,...,m, \forall j = 1,...,n \end{array}\right.$$

Dans cette formulation, la variable x_{ij} est fixée à 1 si l'objet j est affecté au sac i.
Une application possible du problème *SAD-M* est le chargement de cargos (Eilon et Christofides [EC71]). Ce problème est aussi un cas particulier du problème d'affectation généralisée (*PAG*). Dans le *PAG* chaque objet j a un profit c_{ij} au lieu de c_j et les contraintes $\sum_{i=1}^{m} x_{ij} \le 1 \; \forall j = 1,...,n$ sont remplacées par $\sum_{i=1}^{m} x_{ij} = 1 \; \forall j = 1,...,n$.

L'algorithme *MTM* de Martello et Toth [MT81] et l'algorithme *Mulknap* de Pisinger [Pis99a] sont deux méthodes exactes pour résoudre le problème *SAD-M*. Le premier

algorithme repose sur une méthode de séparation et évaluations. Les bornes supérieures sont obtenues à l'aide d'une relaxation surrogate et les bornes inférieures en résolvant *m* problèmes de type *SAD* pris individuellement. Pisinger propose dans le second algorithme différentes améliorations. Il détermine les bornes inférieures en résolvant une série de problèmes de type *SSS*, et utilise la méthode de décomposition d'Horowitz et Sahni [HS74] pour résoudre ces problèmes. Un ensemble d'expériences numériques est présenté pour ces deux approches par Kellerer, Pferschy et Pisinger [KPP04]. L'algorithme *Mulknap* est capable de résoudre des instances fortement corrélées en des temps très raisonnables comprenant jusqu'à 100 000 variables et 10 sacs.

1.3.2 Sac-à-dos avec contraintes disjonctives

Le problème du sac-à-dos avec contraintes disjonctives (*SAD-D*) a été introduit par Yamada et Kataoka [YK01]. Ils modélisent un problème de sélection de sites pour installer des centrales nucléaires. Supposons que nous disposons d'un certain capital d'investissement, noté *b*, et de *n* sites possibles pour construire un ensemble de centrales nucléaires. Chaque site *j* nécessite un coût a_j et permettra de produire une quantité d'énergie c_j. Deux sites choisis doivent obligatoirement être à une distance minimale notée *D* l'un de l'autre (pour des raisons de sécurité et d'environnement). L'objectif du problème consiste à sélectionner un sous-ensemble de sites de manière à maximiser la quantité totale d'énergie produite sans dépasser le capital total à la disposition, et de telle sorte que deux sites soient au moins distants de *D*. Cette contrainte oblige les objets sélectionnés à être compatibles deux à deux. Le programme linéaire suivant décrit ce problème.

$$(SAD\text{-}D) \quad \begin{cases} \max & \sum_{j \in N} c_j x_j \\ s.c. & \sum_{j \in N} a_j x_j \le b \\ & x_j + x_k \le 1 \quad \forall (j,k) \in E \\ & x \in \{0,1\}^n \end{cases}$$

Cette formulation ressemble à celle du problème *SAD*, mais ici un ensemble de contraintes disjonctives est ajouté au problème.

Ainsi, *E* représente l'ensemble des couples incompatibles, c'est-à-dire :

$$E \subseteq \{(j,k) \mid 1 \le j \ne k \le n\} \tag{1.7}$$

Ce problème est un problème NP-difficile, puisque pour $E = \varnothing$ il se réduit au problème *SAD* qui est lui-même NP-difficile. Il est récent et la littérature sur le sujet est

encore assez pauvre. Yamada et Kataoka [YK01] et Yamada, Kataoka et Watanabe [YKW02] ont proposé une heuristique de type glouton ainsi qu'une heuristique basée sur la recherche locale pour générer des bornes inférieures de ce problème. Ils utilisent aussi une relaxation Lagrangienne pour déterminer une borne supérieure du problème, et ils ont finalement développé un algorithme exact basé sur la méthode de séparation et évaluations pour le résoudre.

Hifi et Michrafy [HM05] ont récemment proposé un algorithme basé sur la recherche locale réactive pour résoudre ce problème. Ils introduisent l'autorisation de mouvements dégradants pour sortir des optima locaux et diversifier la recherche, ainsi que l'utilisation de la mémoire pour éviter de visiter plusieurs fois une même solution. Cet algorithme est utilisé dans un autre article par Hifi et Michrafy [HM06] pour fournir une solution initiale de bonne qualité à un algorithme exact permettant de rivaliser avec une version récente du logiciel d'optimisation CPLEX d'Ilog [Ilo03].

1.3.3 Sac-à-dos à choix multiples

Le problème du sac-à-dos à choix multiples (*SAD-CM*) est une extension du problème *SAD* dans laquelle les objets sont partitionnés en différentes classes. Dans chaque classe, un objet et un seul doit être sélectionné. La formulation suivante introduit ce problème

$$(SAD-CM)\quad \begin{cases} \max & \displaystyle\sum_{k=1}^{d}\sum_{j\in N_k} c_{kj}x_{kj} & \\ s.c. & \displaystyle\sum_{k=1}^{d}\sum_{j\in N_k} a_{kj}x_{kj} \le b & \\ & \displaystyle\sum_{j\in N_k} x_{kj} = 1 & k=1,...,d \\ & x_{kj}\in\{0,1\} & k=1,...,d,\ j\in N_k \end{cases}$$

avec d le nombre total de classes et N_k la classe k. En supposant que chaque classe N_k a une taille égale à n_k, le nombre total d'objets est défini par $n=\sum_{k=1}^{d} n_k$.

Plusieurs applications de ce problème peuvent être trouvées dans la littérature. Ainsi Nauss [Nau78] l'a utilisé pour la résolution de problèmes de gestion de budgets, Sinha et Zoltners [SZ79] pour l'allocation de ressources et Tillman, Hwang et Kuo [THK80] pour la fiabilité de systèmes.

Différents algorithmes de séparation et évaluations ont été proposés pour ce problème par Sinha et Zoltners [SZ79], Dyer, Kayal et Walker [DKW84]. Dyer, Riha et Walker

[DRW95] ont proposé un algorithme hybride combinant la programmation dynamique et une méthode de séparation et évaluations pour ce problème. Ils utilisent un dual Lagrangien pour générer des bornes supérieures de bonne qualité à chaque nœud de l'arbre de recherche. Ils utilisent aussi une procédure de réduction pour diminuer la taille du problème original. Ils ont testé leur algorithme sur des instances comportant jusqu'à 200 classes et 20000 variables et les résultats obtenus montrent l'efficacité de l'approche.

Pisinger [Pis95b] a proposé un algorithme appelé *Mcknap* basé sur la programmation dynamique avec l'utilisation d'une liste et sur un problème noyau. Les classes sont ajoutées au problème noyau consécutivement de manière à augmenter la taille de ce problème au cours du processus de résolution, et des règles de réduction sont également utilisées pour fixer des variables à leur valeur optimale.

Il est possible de définir le problème du sac-à-dos multidimensionnel à choix multiples (*SADM-CM*) de la même manière que le problème *SAD-MC* est obtenu à partir du problème *SAD*. Ce problème étant très difficile, la majorité des publications le concernant décrivent des approches heuristiques. Une des premières publications pour le *SADM-MC* est due à Moser, Jokanovic et Shiratori [MJS97]. Ils présentent un algorithme utilisant des multiplicateurs Lagrangiens. Les heuristiques proposées pour résoudre le *SADM-CM* sont jusqu'à présent généralement basées sur des heuristiques existantes pour résoudre le problème *SADM*. Ainsi, Parra-Hernandez et Dimopoulos [PHD02] proposent une extension d'une approche due à Pirkul [Pir87]. Ils relâchent les contraintes de choix multiples dans une première phase. Le problème obtenu consiste à sélectionner au moins un objet par classe au lieu d'un et un seul. Ils calculent ensuite un ensemble de multiplicateurs Lagrangiens à l'aide d'une relaxation Lagrangienne et trient les variables selon leur efficacité. Ils construisent finalement une solution initiale à l'aide d'une approche gloutonne et modifient celle-ci de manière à la rendre réalisable (si elle ne l'est pas déjà) par une recherche locale.

1.3.4 Sac-à-dos multidimensionnel avec contraintes de demande

Le problème du sac-à-dos multidimensionnel avec contraintes de demandes (*SADMD*) est une extension du problème *SADM*. Il consiste à introduire, en plus des contraintes d'inégalité de type $\leq$, une ou plusieurs contraintes d'inégalité de type $\geq$, dites de demande.

Ce problème est apparu dans la littérature sous cette dénomination avec Cappanera [Cap99]. La formulation de ce problème sous la forme d'un programme linéaire est la suivante.

$$(SADMD)\quad \begin{cases} \max & \sum_{j\in N} c_j x_j \\ s.c. & \sum_{j=1}^{n} a_{ij} x_j \le b_i \quad i \in M^1 = \{1,2,...,m_1\} \\ & \sum_{i=1}^{n} a_{ij} x_j \ge b_i \quad i \in M^2 = \{1,2,...,m_2\} \\ & x \in \{0,1\}^n \end{cases}$$

avec M^1 qui désigne les contraintes de capacité de type $\leq$ du problème *SADM*, et M^2 qui représente l'ensemble des contraintes de demande de type $\geq$.

Le fait d'ajouter des contraintes de demande au problème *SADM* complique fortement sa résolution (déjà non triviale). La littérature sur ce problème récent n'est pas encore très importante. Cependant, de nombreuses applications pratiques ont la structure du problème *SADMD* telles que la sélection de projets et de budgets (voir Beaujon, Marin et McDonald [BMM01]), ou encore les problèmes de localisation d'installations indésirables traités par exemple par Cappanera [Cap99], Plastria [Pla01], Romero-Morales, Carrizosa et Conde [MCC97]. Ce problème fait bien sûr partie de la classe des problèmes NP-difficiles, et l'utilisation d'heuristiques est recommandée même pour des instances de taille moyenne.

Une des approches les plus efficaces à ce jour pour résoudre ce problème est due à Cappanera et Trubian [CT05]. Ils définissent une heuristique sophistiquée de type recherche locale en deux phases. La première phase consiste à essayer d'obtenir un ensemble de solutions réalisables du problème. Une phase de recherche locale est ensuite appliquée à partir de chacune d'entre elles, en restant dans le domaine réalisable tout au long du processus. Cette méthode est basée sur différentes idées proposées par Glover et Kochenberger [GK96] dans un contexte de stratégie d'oscillation. Cappanera et Trubian ont aussi généré un ensemble important d'instances disponibles sur Internet [Bea90]. Il est parfois très difficile pour certaines d'entre elles d'obtenir une solution réalisable, surtout lorsque le nombre de contraintes de demande augmente. Ainsi un logiciel performant tel que CPLEX de Ilog rencontre de grandes difficultés pour traiter certaines de ces instances. Hvattum et Lokketengen [HL05] ont récemment proposé différentes implémentations d'un algorithme de recherche dispersée pour ce problème et effectué un ensemble important d'expériences

numériques montrant que cette approche permet d'obtenir des solutions réalisables de bonne qualité pour une grande majorité des instances existantes.

1.4 Le sac-à-dos multidimensionnel avec objectif non linéaire

Nous pouvons également trouver dans la littérature d'autres problèmes obtenus à partir des problèmes *SAD* ou *SADM* en modifiant leur objectif.

1.4.1 Problèmes de la distribution équitable et du sac-à-dos max-min

Le problème de la distribution équitable ou du sac-à-dos partagé (couramment appelé *knapsack sharing problem* en anglais) a été introduit par Brown [Bro79]. Ce problème, noté *PDE*, est défini sur un ensemble N de n variables avec une capacité totale b comme pour le problème *SAD*. A chaque objet j ($j=1,...,n$) est également associé un profit c_j et un poids a_j. L'ensemble des objets est divisé en *nbc* différentes classes comme pour le sac-à-dos à choix multiples. Une fonction linéaire est associée à chaque classe d'objets. Le but du problème est de déterminer un sous-ensemble d'objets à inclure dans le sac en respectant la contrainte de capacité tout en maximisant la valeur minimale d'un ensemble de fonctions linéaires :

$$(PDE)\quad \left[\begin{array}{lll} \max & \min\limits_{1\le i\le nbc} & \left\{\sum\limits_{j\in J_i} c_j x_j\right\} \\ s.c. & & \sum\limits_{j\in N} a_j x_j \le b \\ & & x_j \in \{0,1\} \quad \forall j = 1,...,n \end{array}\right.$$

avec J_i qui représente la classe i, pour $i = 1,\ldots,nbc$. Les conditions suivantes sont également vérifiées.

$$\cup_{i=1}^{nbc} J_i = N\ ,\ J_p \cap J_q = \varnothing \text{ et } \forall p = 1,...,nbc \text{ et } \forall q = 1,...,nbc \text{ avec } p \neq q \qquad (1.8)$$

Ce problème permet de représenter des situations particulières lors de la modélisation de problèmes d'allocations. Les objets peuvent par exemple appartenir à plusieurs propriétaires (chaque propriétaire est représenté par une classe), et chaque propriétaire souhaite maximiser le profit total associé à ses objets en fonction de la capacité totale qui est partagée entre tous les propriétaires. Le modèle maximise alors le niveau minimum des profits qui peut être garanti pour chaque propriétaire.

Yamada, Futakawa et Kataoka [YFK98] ont proposé un algorithme exact basé sur la programmation dynamique et un autre basé sur une méthode de séparation et évaluations pour résoudre le problème *PDE*. Les bornes supérieures nécessaires à l'algorithme de séparation et évaluations sont obtenues par une décomposition du problème et une reformulation en problèmes de type *SAD*, et les bornes inférieures sont calculées à l'aide d'une heuristique de type glouton. Les résultats obtenus montrent que la méthode ne permet pas de résoudre des instances de grandes tailles. L'algorithme basé sur la programmation dynamique utilise l'algorithme d'Horowitz et Sahni [HS74] pour résoudre des problèmes *SAD* avec un objectif formulé sous la forme d'une minimisation. Les résultats obtenus avec cette approche sont meilleurs mais les instances fortement corrélées ne sont généralement pas résolues.

Hifi, M'Halla et Sadfi [HHS05] ont récemment proposé un algorithme exact permettant d'améliorer les résultats obtenus par Hifi et Sadfi [HS02]. L'algorithme repose sur des techniques de programmation dynamique et sur une décomposition du problème en une série de problèmes *SAD*.

Quelques algorithmes approchés existent aussi pour ce problème. Hifi, Sadfi et Sbihi [HSS02] ont par exemple proposé une approche basée sur la métaheuristique tabou et les méthodes de recherche locale pour résoudre ce problème qui permet d'obtenir de bons résultats pour des problèmes corrélés ou non.

Un autre problème proche du *PDE* peut être obtenu d'une manière analogue au problème du sac-à-dos multiobjectifs décrit dans la section 1.4.2. Il s'agit du problème du sac-à-dos max-min (*SAD-MM*). Il est défini à partir du problème *SAD* en introduisant *nbp* profits par variable et ainsi *nbp* objectifs différents dans le problème. Le problème consiste à sélectionner un sous-ensemble d'objets en respectant une contrainte de capacité de telle sorte que le profit total minimum soit maximisé. Plus formellement, le modèle mathématique suivant décrit le problème *SAD-MM*.

$$(SAD\text{-}MM) \quad \begin{cases} \max & \min\limits_{k=1,\ldots,nbp} \sum\limits_{j \in N} c_{kj} x_j \\ s.c. & \sum\limits_{j \in N} a_j x_j \leq b \\ & x \in \{0,1\}^n \end{cases}$$

Ce problème a été introduit par Yu [Yu96] qui donne des applications dans le domaine de la gestion de budgets. Le retour d'investissement associé à un projet peut dépendre du déroulement de *nbp* scénarios différents dans le futur. Un investissement peut être fait s'il maximise le retour minimal sur l'ensemble des scénarios. Yu a montré que lorsque *nbp* est

constant, ce problème peut être résolu en temps pseudo-polynomial à l'aide de la programmation dynamique. Il a aussi proposé un algorithme de type séparation et évaluations pour résoudre ce problème en utilisant des bornes supérieures générées à l'aide de contraintes surrogates.

1.4.2 Sac-à-dos multiobjectifs

Il peut arriver qu'un problème réel consiste à optimiser plusieurs objectifs (ou critères). Nous parlons alors d'optimisation multiobjectifs. Pour illustrer la notion de multiobjectivité, nous présentons ci-dessous le problème du sac-à-dos bi-objectifs (*SAD-BO*) qui comporte deux objectifs à maximiser.

$$(SAD\text{-}BO) \quad \begin{cases} \max & \sum_{j \in N} c_{1j} x_j \\ \max & \sum_{j \in N} c_{2j} x_j \\ s.c. & \sum_{j \in N} a_j x_j \leq b \\ & x \in \{0,1\}^n \end{cases}$$

Nous notons dans la suite par *SAD-MO* le problème du sac-à-dos multiobjectifs, et par *SADM-MO* le problème du sac-à-dos multidimensionnel multiobjectifs. Nous présentons ici quelques notions de base de l'optimisation multiobjectifs avant d'aborder quelques méthodes de résolution du problème *SAD-MO*. Ehrgott et Gandibleux [EG02] présentent de nombreux concepts liés à ce type d'optimisation.

Le fait d'avoir plusieurs critères à optimiser implique qu'il n'est plus possible de définir la notion de solution optimale. L'objectif du problème est ici de trouver l'ensemble des solutions réalisables non dominées, encore appelées solutions *efficaces*, ou *Pareto optimales*. Lorsqu'il y a plusieurs critères, une solution x_1 *domine fortement* une solution x_2, si elle est meilleure sur l'ensemble des critères. En notant *nbc* le nombre de critères, x^1 domine fortement x^2 si les *nbc* inégalités suivantes sont vérifiées.

$$\sum_{j \in N} c_{kj} x_j^1 \geq \sum_{j \in N} c_{kj} x_j^2, \forall k \in \{1,...,nbc\} \tag{1.9}$$

Finalement une solution x^1 domine une solution x^2 si au moins une des *nbc* inégalités de (1.9) est stricte.

Différentes applications existent pour les problèmes *SAD-MO* et *SADM-MO*. Rosenblatt et Sinuany-Stern [RS89] ont abordé des problèmes de gestion de budgets avec

deux objectifs. Le premier objectif est de maximiser le nombre de projets acceptés, et le second consiste à minimiser le risque total associé à ces projets.

Différents algorithmes exacts ont été proposés pour résoudre le problème *SAD-MO*. Eben-Chaime [Ebe96] a ainsi développé une approche basée sur une formulation du problème comme un problème de plus long chemin. Ulungu et Teghem [UT97] ont proposé une approche en deux phases pour construire toutes les solutions efficaces du problème *SAD-BO*.

Gandibleux, Mezdaoui et Fréville [GMF97] ont développé un algorithme de recherche tabou multiobjectifs général pour résoudre les problèmes d'optimisation multiobjectifs. Les éléments associés à la mémoire de l'algorithme tabou sont utilisés pour mettre à jour le poids des variables et diversifier la recherche. Gandibleux et Fréville [GF00] ont appliqué cet algorithme au problème *SAD-BO*. Schaffer [Sch85] a lui développé une méthode basée sur les algorithmes génétiques pour les problèmes multiobjectifs. Il divise la population en plusieurs sous-populations, et un critère à optimiser est associé à chaque sous population. Les règles de sélection sont appliquées à la population globale. Gandibleux, Morita et Katoh [GMK01] ont appliqué un tel algorithme au problème *SAD-MO*.

1.4.3 Sac-à-dos quadratique

Le problème du sac-à-dos quadratique (*SADQ*) consiste à considérer que le profit obtenu en choisissant un objet ne dépend pas seulement de l'objet lui-même, mais aussi des autres objets choisis. Cela permet de modéliser certains problèmes réels en théorie des graphes par exemple. Le problème *SADQ* a été introduit par Gallo, Hammer et Simeone [GHS80] et sa formulation est la suivante.

$$(SADQ) \quad \begin{cases} \max & \sum_{i \in N} \sum_{j \in N} c_{ij} x_i x_j \\ s.c. & \sum_{j \in N} a_j x_j \leq b \\ & x \in \{0,1\}^n \end{cases}$$

Dans ce problème, chaque coefficient c_{jj} est le profit si l'objet j est sélectionné, et $c_{ij} + c_{ji}$ est le profit obtenu si les deux objets i et j sont sélectionnés. Ce problème a été assez largement étudié de par sa structure simple et sa difficulté de résolution importante. L'algorithme *QuadKnap* proposé récemment par Caprara, Pisinger et Toth [CPT99] permet par exemple de résoudre certaines grandes instances de la littérature.

1.5 Conclusions

Nous avons présenté dans ce chapitre quelques problèmes de la famille du sac-à-dos en commençant par le problème « père », très connu et encore très étudié de nos jours. Nous avons décrit un ensemble d'applications existantes pour ce problème et le problème du sac-à-dos multidimensionnel. Les méthodes de résolution existantes pour ce dernier font l'objet du second chapitre de ce mémoire. Nous avons finalement présenté quelques autres problèmes existants obtenus en modifiant les contraintes et/ou l'objectif des problèmes *SAD* ou *SADM*. La présentation de ces problèmes nous a permis de montrer la grande diversité des applications possibles pour les problèmes modélisables sous la forme de problèmes de type sac-à-dos. Ajoutons que d'autres problèmes existent comme le sac-à-dos borné, non borné, ou encore avec des contraintes de précédence par exemple.

Chapitre 2

Méthodes de résolution classiques et applications au sac-à-dos multidimensionnel en variables 0-1

Comme nous avons pu le voir dans le chapitre précédent, les problèmes de la famille du sac-à-dos permettent de modéliser de nombreuses applications, et les références sur le sujet sont importantes. Dans la suite de ce mémoire, nous nous focalisons sur le problème du sac-à-dos multidimensionnel en variables 0-1 (*SADM*). Même si nous avons pu voir que sa formulation est simple et que sa compréhension l'est tout autant, il est clairement établi que ce problème représente toujours de nos jours un challenge intéressant puisque aucune approche ne permet de garantir l'optimalité des solutions obtenues sur des instances de tailles moyennes en des temps raisonnables. C'est entre autres pourquoi nous avons choisi ce problème pour tester les différentes approches que nous avons proposées. Ce choix s'explique aussi par le fait qu'il représente un cas particulier de la programmation en nombres entiers.

Nous présentons dans ce chapitre un ensemble de méthodes existantes pour la résolution de problèmes d'optimisation en nombres entiers et nous insistons sur les approches efficaces proposées pour résoudre le problème *SADM*. Nous commençons par aborder le calcul de bornes supérieures qui peuvent être utilisées dans des algorithmes de séparation et évaluations par exemple. Les méthodes exactes efficaces pour résoudre le problème du sac-à-dos multidimensionnel sont assez peu nombreuses devant sa difficulté de résolution, en particulier lorsque le nombre de contraintes augmente. Nous référençons cependant quelques algorithmes ayant permis l'obtention de bons résultats. Nous abordons ensuite quelques heuristiques efficaces proposées ces dernières années, ainsi que des métaheuristiques. Les références sur ce sujet ne manquent pas, et nous présentons quelques travaux ayant permis d'obtenir des solutions de très bonne qualité et/ou avec un rapport entre la qualité de la solution et le temps d'exécution intéressant. Nous terminons ce chapitre par la présentation de quelques méthodes hybrides qui combinent plusieurs approches de résolution pour essayer de tirer bénéfice des points forts de chacune d'elles.

2.1 Calcul de bornes supérieures et résolution exacte

Lors de la résolution exacte d'un problème d'optimisation, le calcul de bornes supérieures (dans le cas d'un problème de maximisation) est souvent nécessaire. L'obtention d'une borne supérieure et d'une borne inférieure permet également d'avoir un encadrement de la valeur optimale du problème. Cet encadrement est défini comme suit :

$$\underline{v} \leq cx^* \leq \bar{v} \tag{2.1}$$

avec $\underline{v}$ (resp. $\bar{v}$) la valeur d'une borne inférieure (resp. une borne supérieure) du problème, et cx^* la valeur optimale du problème.

Une approche classique en recherche opérationnelle pour obtenir une borne supérieure d'un problème en nombres entiers consiste à relâcher les contraintes d'intégrité. Le programme linéaire *PL*, présenté ci-après, est obtenu à partir de (1.1) (chapitre 1, section 1.1). Il s'agit de la relaxation en continu du problème initial. La valeur optimale de ce problème est nécessairement plus grande que celle du problème original car l'ensemble des solutions réalisables du problème original est un sous-ensemble des solutions réalisables du problème *PL*.

$$(PL) \quad \begin{cases} \max & cx \\ s.c. & Ax \leq b \\ & x \in [0,1]^n \end{cases}$$

Ce problème en variables continues peut par exemple être résolu en appliquant l'algorithme du simplexe de Dantzig [Dan57]. Même si la complexité théorique de cet algorithme est exponentielle dans le pire des cas, l'utilisation pratique de celui-ci permet de résoudre rapidement la plupart des problèmes de taille raisonnable.

La résolution du problème *PL* est généralement rapide pour obtenir une borne supérieure du problème *SADM*. La qualité de cette borne supérieure n'est cependant pas toujours très bonne, et elle dépend du saut de dualité du problème, c'est-à-dire de l'écart entre la valeur de cette borne et la valeur optimale du problème. Il existe d'autres types de relaxations plus coûteuses mais plus précises. Il s'agit des relaxations Lagrangienne, surrogate, et composite principalement.

2.1.1 Relaxation Lagrangienne

Le problème du sac-à-dos multidimensionnel peut être reformulé comme suit.

$$(SADM') \quad \begin{bmatrix} \max & cx \\ s.c. & A^1 x \le b^1 \\ & A^2 x \le b^2 \\ & x \in \{0,1\}^n \end{bmatrix}$$

avec A^1 une matrice de dimension $m^1 \times n$, A^2 une matrice de dimension $m^2 \times n$, avec $m = m^1 + m^2$. De même, b^1 et b^2 sont respectivement de dimension m^1 et m^2.

Nous supposons dans cette formulation que les contraintes $A^1x \le b^1$ sont les contraintes dites *faciles* du problème, et les contraintes $A^2x \le b^2$ sont les contraintes dites *difficiles*. L'idée sur laquelle repose la relaxation Lagrangienne consiste à intégrer les contraintes difficiles dans l'objectif du problème avec une certaine pénalité pour générer un problème plus facile à résoudre. La relaxation Lagrangienne ($LR(\lambda)$) du problème *SADM* associée aux contraintes $A^2x \le b^2$ est ainsi définie par :

$$(LR(\lambda)) \quad \begin{bmatrix} \max & cx & + & \lambda(b^2 - A^2 x) \\ s.c. & A^1 x & \le b^1 & \\ & x \in \{0,1\}^n & & \end{bmatrix}$$

avec $\lambda \in IR_+^{m^2}$.

Les contraintes difficiles du problème n'apparaissent ainsi plus comme contraintes, et le terme $\lambda(b^2 - A^2x)$ assure le fait d'avoir une pénalité négative (car λ est positif).

Le dual Lagrangien (L) associé est obtenu comme suit.

$$(L) \quad \min_{\lambda \ge 0} v(LR(\lambda))$$

Cette relaxation a été prouvée comme étant un outil efficace pour la résolution de problèmes en nombres entiers (Fischer [Fis81]). La valeur du dual Lagrangien est toujours au moins aussi bonne que la valeur de la relaxation en continu du problème *SADM* :

$$v(L) \le v(PL) \tag{2.2}$$

Lorsque $m^2 = m$, les deux relaxations ont la même valeur.

Notons pour terminer que nous nous sommes volontairement placés dans le cas du problème *SADM* mais que les propriétés précédentes restent vraies pour la résolution des problèmes d'optimisation.

2.1.2 Relaxation surrogate

La relaxation surrogate a été proposée par Glover dans les années 1960 [Glo65, Glo68]. Elle consiste à remplacer l'ensemble des contraintes du problème par une seule, dite contrainte surrogate. Une relaxation surrogate du problème *SADM* est définie comme suit.

$$(SR(\mu)) \quad \begin{cases} \max & \sum_{j\in N} c_j x_j \\ s.c. & \sum_{j\in N}\left(\sum_{i\in M}\mu_i a_{ij}\right)x_j \leq \sum_{i\in M}\mu_i b_i \\ & x\in\{0,1\}^n \end{cases}$$

avec $\mu \geq 0$. Ce type de relaxation ramène la résolution du problème *SADM* à la résolution d'un problème *SAD*.

Rappelons que si une solution optimale de $SR(\mu)$ est réalisable pour le problème *SADM*, alors cette solution est optimale pour ce problème.

Le dual surrogate (*S*) est :

$$(S) \qquad \min_{\mu\geq 0} v(SR(\mu))$$

Greenberg et Pierskalla [GP70] ont montré que la valeur du dual surrogate est toujours au moins aussi bonne que la valeur de la relaxation en continu.

$$v(S) \leq v(PL) \tag{2.3}$$

2.1.3 Relaxation composite

Une dernière relaxation courante qui combine les deux relaxations précédentes est la relaxation composite proposée par Greenberg et Pierskalla [GP70]. Une relaxation composite pour le problème *SADM*, associée à λ et μ deux vecteurs de multiplicateurs réels, est définie comme suit.

$$CR(\lambda,\mu) \quad \begin{cases} \max & \sum_{j\in N}\left(c_j - \sum_{i\in M}\lambda_i a_{ij}\right)x_j & + & \sum_{i\in M}\lambda_i b_i \\ s.c. & \sum_{j\in N}\left(\sum_{i\in M}\mu_i a_{ij}\right)x_j & \leq & \sum_{i\in M}\mu_i b_i \\ & x \in \{0,1\}^n & & \end{cases}$$

Le dual composite (*C*) est obtenue par :

$$(C) \quad \min_{\lambda\geq 0,\mu\geq 0} v(CR(\lambda,\mu))$$

Les relaxations Lagrangienne et surrogate sont des cas particuliers de la relaxation composite. La valeur du dual composite est également toujours au moins aussi bonne que la valeur de la relaxation en continu.

$$v(C) \leq v(PL) \tag{2.4}$$

Un résultat théorique prouvé par Crama et Mazzola [CM94] permet d'avoir une idée sur l'amélioration limite que peut apporter la relaxation composite (ainsi que les relaxations lagrangienne et surrogate) par rapport à la relaxation en continu. Cette amélioration est au plus égale à la valeur du plus grand coefficient c_j dans l'objectif pour le problème *SADM*.

Finalement, le théorème suivant permet une comparaison des résultats obtenus entre les quatre relaxations.

Théorème 2.1 :

$$v^* \leq v(C) \leq v(S) \leq v(L) \leq v(PL) \tag{2.5}$$

avec *v** la valeur optimale du problème.

2.1.4 Méthodes de résolution exacte

Les algorithmes exacts les plus efficaces pour résoudre le problème *SADM* sont essentiellement de type séparation et évaluations.

Les premières références de Cabot [Cab70] et Thesen [The75] ont permis l'obtention de résultats moyens. Le premier algorithme est basé sur la méthode d'élimination de Fourier-Motzkin, alors que le second repose sur un principe récursif dans lequel l'auteur portait une attention particulière au gain d'espace mémoire.

Une approche plus élaborée est due à Shih [Shi79]. Il prend en compte la structure particulière du problème *SADM*. Le calcul des bornes supérieures à chaque nœud de l'arbre de recherche et les règles de branchement dépendent en effet de l'information générée par la

relaxation en continu des *m* problèmes de sac-à-dos pris séparément. Les deux points faibles de cet algorithme sont son coût important en mémoire et son incapacité à résoudre des problèmes avec des contraintes de capacité serrée.

Un des algorithmes de type séparation et évaluations les plus efficaces pour résoudre le problème *SADM* a été proposé par Gavish et Pirkul [GP85]. Les auteurs utilisent la relaxation surrogate et différentes règles de réduction pour diminuer la taille du problème avant d'appliquer l'algorithme de séparation et évaluations. Les résultats obtenus sont de bonne qualité, et les auteurs montrent en particulier que leur algorithme est nettement plus rapide que l'algorithme de Shih.

Plusieurs auteurs ont également présenté des algorithmes basés sur la programmation dynamique pour résoudre le problème *SADM* de manière optimale comme Gilmore et Gomory [GG66], Weingartner et Ness [WN67]. Ces derniers ont proposé un algorithme *dual*. L'approche consiste à partir d'une solution incluant tous les objets. Des objets sont ensuite retirés de la solution au cours de la phase de programmation dynamique jusqu'à ce qu'une solution réalisable soit rencontrée. Des concepts de réduction à l'aide de bornes inférieure et supérieure sont également présentés pour améliorer le processus.

Notons pour terminer que Frieze et Clarke [FC84] ont proposé des schémas d'approximation polynomiaux pour le problème *SADM*. Il n'existe cependant pas de schéma d'approximation totalement polynomial pour ce problème (à moins que $P = NP$) selon Gens et Levner [GL79] et Korte et Schrader [KS81].

2.2 Heuristiques

De nombreuses heuristiques peuvent être référencées pour la résolution des problèmes d'optimisation et en particulier pour le problème *SADM*. Certaines d'entre elles sont proposées pour la résolution d'un problème particulier lorsqu'elles prennent en compte des caractéristiques propres à celui-ci. Nous commençons cette section par la présentation du principe des heuristiques de type glouton. Nous abordons ensuite quelques heuristiques basées sur des relaxations.

2.2.1 Heuristiques de type glouton

Les heuristiques de type glouton sont généralement simples à programmer, et requièrent un faible temps d'exécution pour générer une solution réalisable du problème. En

partant d'une solution nulle (resp. avec toutes les variables fixées à 1), le principe de ces heuristiques est d'ajouter (resp. retirer) des éléments dans la solution courante tant que (resp. jusqu'à ce que) la solution reste (resp. devienne) réalisable dans le cas des heuristiques gloutonnes dites primales (resp. duales).

Une des questions majeures qui se pose est de savoir dans quel ordre il faut considérer les variables. Une façon de faire consiste à trier les variables selon l'ordre décroissant (resp. croissant) des rapports suivants dans le cas primal (resp. dual) pour le problème *SADM*.

$$\frac{c_j}{\sum_{i \in M} w_i a_{ij}} \tag{2.6}$$

avec $w = (w_1, w_2, \ldots, w_m)$ un vecteur de m poids positifs.

Ce rapport définit ce qui est souvent référencé comme l'efficacité des variables.

Senju et Toyoda [ST68] ont proposé une des premières heuristiques de type glouton pour le problème *SADM*. Cet algorithme fait partie de la classe des heuristiques duales. Plusieurs approches primales ont également été proposées par Kochenberger, McCarl et Wymann [KMW74], Toyoda [Toy75] ou Loulou et Michaelides [LM79] par exemple. Ces derniers ont étendu et amélioré l'heuristique de Toyoda en intégrant entre autres un calcul itératif du multiplicateur w au cours du processus.

D'autres algorithmes plus sophistiqués sont ensuite apparus. Ainsi, Magazine et Oguz [MO84] ont couplé l'heuristique de Senju et Toyoda avec l'utilisation d'une relaxation Lagrangienne pour fixer des variables à leur valeur optimale. Fréville et Plateau [FP86] ont également proposé plusieurs algorithmes utilisant la relaxation surrogate et permettant d'accélérer le processus de fixation. Citons pour terminer l'algorithme de Pirkul [Pir87] utilisant une méthode de descente pour déterminer le multiplicateur surrogate. Les résultats obtenus par cet algorithme étaient à l'époque du même ordre qu'une heuristique basée sur la relaxation en continu et réputée comme une approche performante. Il s'agit de l'heuristique du pivot et complément de Balas et Martin [BM80] que nous présentons dans la section suivante.

2.2.2 Heuristiques basées sur des relaxations

Différentes heuristiques basées sur des relaxations permettent d'obtenir de bonnes solutions pour les problèmes en nombres entiers et pour le problème *SADM* en particulier. Nous commençons ici par présenter des heuristiques simples consistant à arrondir une

solution optimale de la relaxation en continu. Nous abordons ensuite des heuristiques plus perfectionnées basées sur la notion de pivot.

2.2.2.1 Relaxation en continu

Une heuristique simple permettant de générer rapidement une solution (réalisable ou non) du problème consiste à arrondir une solution optimale de la relaxation en continu. L'idée derrière cette approche est qu'une solution optimale de la relaxation en continu peut être proche d'une solution optimale du problème. Même si cela est vrai dans certains cas, il n'y a cependant pas de garantie d'obtenir une solution réellement proche de l'optimum dans tous les cas comme nous le montrons dans la suite sur un exemple simple.

Deux types d'approche peuvent être envisagés. Dans le premier cas la solution de la relaxation en continu est *tronquée* de manière à générer une solution réalisable du problème (dans le cas du problème *SADM*). Dans le second cas la solution est *arrondie* (au sens large, c'est-à-dire en fixant les variables fractionnaires aux valeurs entières les plus proches), ce qui amène dans certains cas à une solution non réalisable. Il est également possible de choisir une méthode d'arrondi, et ensuite d'appliquer un opérateur adapté au problème considéré pour transformer la solution obtenue en solution réalisable si elle ne l'est pas.

Ce type d'heuristiques peut être utilisé comme solution initiale dans un algorithme plus complet. Ainsi Salkin [Sal70] utilise par exemple une solution tronquée de la relaxation en continu comme solution de départ d'un algorithme d'énumération implicite. Un avantage de ces heuristiques est qu'elles permettent d'obtenir une solution réalisable très rapidement. La qualité de cette solution n'est cependant généralement pas très bonne, en particulier pour les problèmes difficiles.

L'exemple suivant illustre sur une instance du problème du sac-à-dos multidimensionnel les deux façons de procéder. Cet exemple montre que la méthode de troncature peut amener à une solution très éloignée de la valeur optimale du problème, alors que la méthode d'arrondi peut conduire à une solution non réalisable proche de l'optimum et donc atteignable rapidement.

Exemple 2.1 :

$$(GK_1)\quad \begin{bmatrix} \max & 20x_1 + & 18x_2 + & 15x_3 + & 14x_4 + & 12x_5 + & 9x_6 + & 7x_7 + & 5x_8 + & 3x_9 + & 2x_{10} & \\ s.c. & 15x_1 + & 16x_2 + & 12x_3 + & 12x_4 + & 10x_5 + & 10x_6 + & 8x_7 + & 5x_8 + & 4x_9 + & 3x_{10} & \leq 45 \\ & 22x_1 + & 21x_2 + & 16x_3 + & 14x_4 + & 15x_5 + & 7x_6 + & 5x_7 + & 2x_8 + & 4x_9 + & 4x_{10} & \leq 50 \\ & 18x_1 + & 20x_2 + & 15x_3 + & 10x_4 + & 9x_5 + & 8x_6 + & 2x_7 + & 6x_8 + & 2x_9 + & 5x_{10} & \leq 40 \\ & x_j \in \{0,1\} & j = 1,...,10 & & & & & & & & & \end{bmatrix}$$

La valeur optimale de ce problème vaut $v(GK_1) = 50$. La valeur de la relaxation en continu vaut $v(PL) = 51.60$. Une solution optimale de la relaxation en continu est donnée par : $\bar{x}$ = (0.90 0 0 1 0.58 0.07 1 1 0 0). Si nous appliquons une heuristique d'arrondi à cette solution, nous obtenons la solution x^1 suivante : x^1 = (1 0 0 1 1 0 1 1 0 0), de valeur $cx^1 = 58$, qui n'est pas réalisable mais de valeur assez proche de la valeur optimale. Si nous appliquons cette fois une troncature de la solution $\bar{x}$, nous obtenons la solution x^2 suivante : x^2 = (0 0 0 1 0 0 1 1 0 0), de valeur $cx^2 = 26$, réalisable mais très éloignée de la valeur optimale.

D'autres approches utilisant la relaxation en continu ont aussi été proposées. Hillier [Hil69] a proposé un algorithme en trois phases. La première phase consiste à construire un chemin ayant pour origine une solution optimale de la relaxation en continu et se déplaçant de proche en proche vers le domaine réalisable. Dans une seconde phase le processus poursuit ce chemin de manière à trouver une meilleure solution réalisable et dans une dernière phase une recherche locale est appliquée pour essayer d'améliorer la solution. Balas et al. [BCDMP01] ont développé une heuristique appelée *Octane* pour résoudre les problèmes en variables 0-1. Elle consiste à générer un ensemble de solutions réalisables en déterminant les intersections de facettes associées à un octaèdre contenant la solution de la relaxation en continu.

Zhang et Ong [ZO04] ont eux proposé une heuristique basée sur la relaxation en continu pour le problème *SAD* bi-objectifs. La relaxation en continu est utilisée pour diviser l'ensemble des variables en trois sous-ensembles associés aux indices des variables égales à 0, 1 et des variables fractionnaires d'une solution optimale de cette relaxation. Le principe de l'heuristique pour générer une solution non dominée consiste à fixer une partie des variables fractionnaires à 1 en laissant les autres à 0. Différents moyens sont utilisés pour gérer le nombre de variables sélectionnées à chaque étape. Le processus est répété un certain nombre de fois pour obtenir le plus de solutions non dominées possible.

Différents algorithmes se basant sur des mouvements dits de type pivot ont été proposés au cours des dernières décennies. Ces algorithmes reposent sur la méthode du

simplexe et consiste à modifier la solution courante pour obtenir une solution réalisable du problème. L'article référence de ces approches est dû à Balas et Martin en 1980 [BM80].

2.2.2.2 Heuristique pivot et complément

L'heuristique pivot et complément (*P&C*) a été proposée pour la résolution des problèmes en variables 0-1. Elle permet généralement l'obtention de solutions entières réalisables du problème. Le principe de l'heuristique repose sur le fait que le problème défini par (1.1) est équivalent au programme linéaire suivant.

$$(P\text{-}C)\quad \begin{cases} \max & cx & \\ s.c. & Ax + s = b & \\ & 0 \le x_j \le 1 & j \in N \\ & s_i \ge 0 & i \in M \\ & s & \text{variables en base} \end{cases}$$

Cette heuristique se base également sur la relaxation en continu. Une solution optimale de la relaxation en continu sert de solution initiale et des mouvements sont effectués à partir de cette solution pour essayer d'obtenir une solution entière réalisable. Les mouvements consistent à faire une séquence de pivots de manière à avoir toutes les variables d'écart s_i en base et donc obtenir une solution entière du problème initial. Pour cela la méthode se décompose en deux phases : une phase de recherche et une phase d'amélioration.

Dans la phase de recherche, l'heuristique essaie de trouver un mouvement pivot qui réduise l'écart avec la frontière réalisable. Trois types de pivots sont définis avec un ordre de priorité. La phase de recherche s'achève lorsqu'une solution entière réalisable est obtenue ou lorsque la réalisabilité est définitivement perdue. Dans ce cas l'heuristique aboutit à un échec, aucune solution réalisable n'a pu être obtenue pour le problème.

La phase d'amélioration débute dès qu'une solution réalisable est visitée. Le principe consiste à complémenter un sous-ensemble de variables entières dans le but d'obtenir la plus grande amélioration possible de la solution courante. Un mouvement correspond dans un premier temps à complémenter une seule variable à la fois. S'il n'y a pas d'amélioration, deux variables sont complémentées simultanément, et finalement trois s'il n'y a toujours pas d'amélioration. Notons que ce dernier type de complément sur trois variables est souvent écarté lors des implémentations de la méthode pour des raisons de complexité.

Les auteurs utilisent également une technique de fixation de variables qui permet dans certains cas de réduire la taille des instances considérées. Finalement, à certaines itérations de

l'algorithme, la procédure tente d'obtenir une solution entière réalisable en appliquant des processus d'arrondi ou de troncature des variables non entières.

Les résultats obtenus sur une variété importante de problèmes sont intéressants. La moitié des problèmes testés à l'époque étaient résolus de manière optimale. Le nombre total de pivots et de compléments est généralement inférieur à $2 \times m$, ce qui semble tout à fait raisonnable. Les résultats obtenus sur les instances de problèmes de sac-à-dos multidimensionnel sont bien meilleurs que ceux de l'heuristique de Toyoda [Toy75] qui était considéré à l'époque comme une heuristique performante pour ce problème. Finalement l'approche peut aussi servir pour générer une solution réalisable qui sert ensuite de point de départ à un algorithme de type séparation et évaluations. Les résultats mentionnés dans ce cas montrent une amélioration en terme de nombre de nœuds nécessaires à l'obtention d'une solution optimale du problème et pour prouver l'optimalité de cette solution. S'ensuit naturellement une amélioration du temps d'exécution par rapport à l'application d'une méthode de séparation et évaluations seule. Cette approche a rencontré un succès important dans les années qui ont suivi sa parution. Ainsi, Marsten et Singhal [MS89] en ont développé une implémentation réutilisable et l'algorithme *P&C* a été intégré dans plusieurs logiciels d'optimisation. Différents auteurs ont proposé des algorithmes utilisant l'heuristique *P&C* avec la recherche tabou. Nous présentons ces approches dans la section 2.3.

Une extension de *P&C* aux problèmes en nombres entiers mixtes a également été proposée par Balas, Schmieta et Wallace [BSW04]. Cette approche combinée avec un logiciel d'optimisation permet d'améliorer les performances de ce logiciel en termes de temps d'exécution et de qualité de la solution. Par rapport à *P&C*, un voisinage de petite taille est défini autour de la solution courante lorsqu'aucun des mouvements de type pivot ne peut être appliqué. La phase d'amélioration de *P&C* est également adaptée de manière à prendre en compte les variables non binaires.

Notons pour terminer l'algorithme de Nediak et Eckstein [NE01] basé sur *P&C* et des méthodes de recherche locale pour résoudre des problèmes en nombres entiers mixtes. L'heuristique repose sur une méthode d'arrondie effectuée à l'aide de pivots et l'utilisation d'une fonction de mesure pour évaluer l'intérêt de choisir un pivot par rapport à un autre. Aucun résultat concernant des instances de problèmes *SADM* n'est cependant répertorié.

2.3 Métaheuristiques

De nombreuses métaheuristiques ont été et sont encore proposées pour résoudre le problème *SADM*. Nous commençons cette section par une présentation de la recherche locale, suivie de quelques références sur la recherche tabou. Nous continuons ensuite en présentant d'autres métaheuristiques.

2.3.1 Recherche locale

La recherche locale (*RL*) est une méthode classique en recherche opérationnelle qui consiste à explorer à une itération donnée le *voisinage* de la solution courante. L'exploration se fait en modifiant un ensemble de composantes de la solution courante pour se déplacer vers une nouvelle solution. Le processus est répété itérativement jusqu'à ce qu'un critère d'arrêt choisi soit rencontré. Ce type d'approche nécessite la définition de plusieurs concepts : celui de *voisinage* associé à une solution, et celui de *mouvement* effectué entre deux itérations.

Le voisinage d'une solution est défini en fonction du problème à résoudre. Il représente, par définition, l'ensemble des solutions accessibles depuis la solution courante x. Le voisinage de x est noté $V(x)$. Un exemple classique de voisinage fréquemment utilisé dans le cas des problèmes en variables binaires est défini ci-dessous par V_1.

$$x' \in V_1(x) \Leftrightarrow \delta(x, x') = 1 \tag{2.7}$$

Dans cette formulation, $\delta(x, x')$ représente le nombre de composantes qui diffèrent entre les solutions x et x' (i.e. $\delta(x, x') = \sum_{j \in N} |x_j - x'_j|$). Une solution x' est voisine d'une solution x selon V_1 lorsqu'une seule composante diffère entre les deux solutions. La taille de ce voisinage est constante quelle que soit la solution x, et est égale à $|V_1(x)| = n$.

Lorsqu'un voisin x' d'une solution x est choisi parmi tous les voisins possibles, la recherche effectue un mouvement de x vers x'. Le type de mouvement défini par le voisinage V_1 est appelé 1-échange, car il ne modifie qu'une composante. Il est possible de définir d'autres voisinages en permettant la modification de k composantes. Ce type de mouvement est alors appelé k-échange.

Hansen et Mladenovic [HM01] ont proposé la métaheuristique appelée *Recherche à voisinage variable* (*RVV*). Dans cette approche, le voisinage courant de la solution est modifié à chaque itération. Cela implique la définition de plusieurs voisinages utilisables pour le problème. La mise en pratique du changement de voisinage peut demander le réglage de

différents paramètres. Contrairement aux autres méthodes basées sur la recherche locale, la *RVV* consiste à explorer des voisinages distants de la solution courante et permet ainsi de définir plusieurs directions de recherche. Chaque application de la méthode de recherche locale permet de générer une solution localement optimale dans le voisinage courant. Il est également possible de changer de voisinage au cours de la phase de recherche locale elle-même car une solution localement optimale dans un voisinage ne l'est pas nécessairement dans un autre voisinage.

Le choix du voisin vers lequel se déplacer dans une méthode classique de recherche locale peut s'effectuer selon différents critères. Il est possible de choisir un voisin aléatoirement dans le voisinage de la solution courante. Cette stratégie, même si elle est tout à fait envisageable, ne représente généralement pas une façon pertinente d'explorer l'espace de recherche dans la majorité des cas car elle ne prend pas en compte les données du problème, ni même les informations associées aux solutions candidates. Une *fonction d'évaluation* est souvent définie de manière à pouvoir comparer les différents mouvements possibles, et choisir celui qui semble être le *meilleur* :

(a) celui ayant la plus grande valeur selon l'objectif du problème ;
(b) celui qui permet de diriger la recherche vers la zone la moins souvent explorée ;
(c) celui qui modifie le plus les capacités résiduelles de la solution courante ;
(d) celui qui est le plus proche de la frontière entre le domaine réalisable et le domaine non réalisable ; *etc*

Notons que dans le cas décrit par (c) les contraintes du problème sont intégrées dans la fonction d'évaluation. Il est également possible d'autoriser la visite de solutions non réalisables au cours de la recherche. Dans ce cas il est nécessaire d'introduire un mécanisme permettant de ne pas trop s'éloigner du domaine des solutions réalisables comme dans (d). La fonction d'évaluation la plus fréquente reste celle décrite par (a). De manière générale, si la stratégie choisie est de type meilleur voisin, le choix de celui-ci à l'itération courante peut alors être formulé de la manière suivante :

$$\text{Trouver } \tilde{x} \in V(x) \text{ tel que } \forall x' \in V(x),\ f(\tilde{x}) \geq f(x') \tag{2.8}$$

avec f qui désigne la fonction d'évaluation utilisée. Par exemple pour (a) $f(\tilde{x})$ peut être remplacée par $c\tilde{x}$.

Le choix de la solution à visiter parmi toutes les solutions candidates se portera donc dans ce cas sur celle qui va maximiser la valeur de la fonction objectif, le but étant de toujours se déplacer vers une solution améliorante.

Un inconvénient majeur de la recherche locale est qu'elle amène souvent le processus rapidement dans un *optimum local*. Etant donnés un voisinage V et une fonction d'évaluation f, un optimum local est défini comme étant un point x de l'espace de recherche vérifiant :

$$\forall x' \in V(x), f(x') \leq f(x) \tag{2.9}$$

Ce point entraîne une situation de blocage puisqu'il n'est plus possible de trouver une solution dans le voisinage de x permettant de faire avancer la recherche. Une idée simple pour éviter ce phénomène consiste à choisir le mouvement qui permet d'obtenir une valeur de f la plus grande possible, tout en autorisant une diminution de la valeur. Ce choix risque cependant d'amener la recherche à générer un cycle autour du point x. La recherche tabou permet d'éviter cette situation grâce à l'introduction de plusieurs composants.

2.3.2 Recherche tabou et applications au problème *SADM*

La recherche tabou (*RT*) a été proposée simultanément par Glover [Glo86] et Hansen [Han86] au milieu des années 1980. Cette approche est une extension des méthodes de recherche locale. L'appellation recherche avec tabou est parfois rencontrée en français mais nous préférons conserver le nom original. Plusieurs notions importantes sont intégrées dans un algorithme de recherche tabou : l'intensification, la diversification, les critères d'aspiration, la liste tabou, *etc*.

Cette métaheuristique est de nos jours très connue, et nous pouvons dire qu'elle rencontre un grand succès auprès des chercheurs en optimisation combinatoire étant donné le nombre conséquent d'articles la référençant. Glover et Laguna [GL97] ont réalisé un ouvrage très détaillé sur cette métaheuristique.

Une des différences majeures de la *RT* par rapport à la *RL* est qu'elle permet au processus de sortir des optima locaux pour continuer l'exploration de l'espace de recherche et aboutir si possible à un optimum global. Un mouvement dégradant par rapport à la fonction d'évaluation peut être choisi à une itération donnée comme dans la recherche locale. L'utilisation d'une *liste tabou* permet cependant d'éviter à la recherche de générer un cycle.

Il existe différents types de listes tabou, et nous pouvons décomposer les méthodes existantes en deux grandes catégories : les listes strictes et les listes non strictes. Dans une liste tabou stricte, toutes les configurations rencontrées durant la recherche doivent être

mémorisées de manière à éviter la visite d'une même solution plusieurs fois. Cela a nécessairement un coût important, quelle que soit la méthode utilisée pour l'implémenter. La méthode d'élimination inverse (MEI) proposée par Glover [Glo90] permet de définir de manière exacte le statut d'un mouvement (savoir s'il est tabou ou non). Le principe général de la MEI est de mémoriser dans une liste les attributs des mouvements effectués, et de construire une autre liste pour générer les mouvements tabous de l'itération courante. Une des premières références pour le problème *SADM* basée sur la recherche tabou a été l'algorithme de Dammeyer et Voss [DV93]. Les auteurs utilisent une méthode de type MEI. Ils montrent que ce type d'approche permet d'obtenir de meilleurs résultats qu'un algorithme de recuit simulé pour ce problème proposé à la même période. Hanafi et Fréville [HF01] ont proposé une extension de la MEI permettant de réduire la complexité de cette approche et obtenir une gestion dynamique de la liste tabou. Dans ce cadre, Battiti et Tecchioli [BT94] ont aussi proposé une autre méthode pour implémenter une liste stricte, la recherche tabou réactive, dans laquelle la taille de la liste tabou est adaptée dynamiquement en fonction de l'historique de la recherche. Différentes implémentations de cette méthode à l'aide d'arbre binaire ou de table de hachage sont comparées sur plusieurs problèmes, et les résultats obtenus sont intéressants en comparaison à des algorithmes génétiques ou de recuit simulé. Le principal inconvénient de cette approche est l'augmentation de la complexité spatiale induite par les structures mises en œuvre.

Une liste non stricte est plus souple et plus facile à implémenter. Le principe consiste à interdire le déplacement de la recherche dans une direction pendant un certain nombre d'itérations. Cette méthode peut permettre d'éviter la visite de solutions déjà visitées mais elle peut ne pas détecter un phénomène de cycles. Nous avons implémenté ce type de liste dans une approche présentée dans le chapitre 3. Notre choix s'est porté sur ce genre de liste essentiellement pour sa simplicité de mise en œuvre, et sa faible complexité comparée aux méthodes précédemment citées. Une des difficultés majeures associées à une liste tabou non stricte est le réglage de la taille de la liste, c'est-à-dire le nombre d'itérations pendant lequel un mouvement est considéré comme tabou. Ce réglage se fait généralement de manière expérimentale.

L'utilisation de la liste tabou peut cependant entraîner un phénomène de blocage dans certaines situations. Il n'y a alors plus aucun mouvement possible et la recherche est confinée dans une solution. La définition de *critères d'aspiration* permet de sortir de cette situation. Un critère d'aspiration consiste à lever le statut tabou d'un mouvement lorsque certaines circonstances sont respectées. Le critère d'aspiration le plus classique consiste simplement à

autoriser le mouvement le moins tabou, c'est-à-dire celui qui aurait pu être fait le plus tôt dans le temps. Un autre critère d'aspiration classique consiste à autoriser un mouvement tabou si celui-ci permet d'améliorer la valeur de la meilleure solution rencontrée depuis le début de la recherche.

Deux autres éléments importants sont généralement associés à la recherche tabou. Il s'agit de la définition de phases d'intensification et de diversification. Une phase d'intensification consiste à effectuer une recherche « approfondie » d'une petite partie de l'espace de recherche. Nous entendons par le terme approfondie le fait que la recherche doit permettre si possible d'explorer de manière optimale la partie de l'espace de recherche concernée. Cela reste difficile à mettre en pratique dans la majorité des cas devant la taille de cette zone qui reste importante. Nous proposons dans le chapitre 3 un processus d'intensification globale grâce auquel nous pouvons explorer une partie de l'espace de recherche de manière exacte. Une phase d'intensification est souvent appliquée à partir d'une solution de bonne qualité de manière à ce que la zone explorée soit « attrayante ».

Le but d'une phase de diversification est au contraire de diriger la recherche dans une partie de l'espace non encore ou peu explorée. Grâce à ce genre de méthode le processus évite de toujours rester dans une partie attractive de l'espace de recherche. La diversification repose souvent sur d'autres structures basées sur la mémoire de la recherche. Ainsi, Glover et Laguna [GL97] parlent de fréquence de mouvements par attribut. Différents types de mémoires dits à court et à long terme peuvent être définis. Nous reviendrons sur ces aspects dans le chapitre 3.

Le problème *SADM* est un problème particulier pour lequel il est connu que les meilleures solutions se situent souvent à la frontière entre les domaines réalisable et non réalisable. Des algorithmes de *RT* efficaces proposés pour ce problème utilisent cette propriété. Ils consistent à autoriser le déplacement de la recherche dans l'espace des solutions non réalisables et ainsi à créer une stratégie d'oscillation entre les deux domaines.

Glover et Kochenberger [GK96] ont défini dans ce cadre un algorithme tabou reposant entre autres sur l'utilisation d'une mémoire dite de récence et d'une fréquence. Leur algorithme alterne des phases constructives et destructives dans un schéma d'oscillation et la recherche est guidée des deux côtés de la frontière à l'aide d'un paramètre définissant le nombre de pas à faire. Cet algorithme a permis l'obtention de très bonnes solutions pour le problème *SADM*. Hanafi et Fréville [HF98] ont proposé une approche tabou permettant d'obtenir des résultats compétitifs vis-à-vis de cet algorithme. La méthode repose sur

l'utilisation d'heuristiques gloutonnes guidées par des informations générées à l'aide de contraintes de type surrogate. Battiti et Techiolli définissent également dans [BT94] des règles autorisant les solutions à violer les contraintes de capacité en introduisant une pénalité dans l'objectif du problème. Le problème obtenu est alors un problème de sac-à-dos non contraint.

Les solutions peuvent également ne pas être réalisables vis-à-vis des contraintes d'intégrité. Ce genre d'approche a souvent été proposé dans un cadre plus large que pour la résolution du problème *SADM*. Aboudi et Jörnsten [AJ94] ont proposé un algorithme tabou pour les problèmes en nombres entiers et en variables 0-1 basé sur la méthode de pivot et complément qui est utilisée comme boîte noire et qui permet d'obtenir des optima locaux. Les auteurs décrivent plusieurs heuristiques qui permettent d'introduire les concepts d'aspiration, de diversification, d'intensification, et de liste tabou dans le processus global. La méthode obtenue permet une amélioration des performances de *P&C*. A la même période, Lokketangen, Jörnsten et Storoy [LJS94] proposent d'intégrer la recherche tabou dans *P&C*. A la différence de la méthode précédente, la recherche tabou est ici insérée à l'intérieur de *P&C*. L'intégration est faite dans les deux phases de *P&C* mais les résultats montrent que c'est l'ajout dans la phase d'amélioration qui permet le plus d'optimiser les résultats de *P&C*. Finalement, citons les travaux de Lokketangen et Glover [LG96, LG98] dans lesquels ils utilisent une méthode basée sur le simplexe comme sous-programme pour effectuer des mouvements de type pivot. Les éléments liés à la mémoire dans la recherche tabou sont intégrés de manière à exclure ou pénaliser certains mouvements.

Vasquez et Hao [VH01b] et Vasquez et Vimont [VV05] ont proposé les algorithmes hybrides basés sur la recherche tabou les plus performants pour le problème *SADM* à ce jour. Dans le premier article les auteurs introduisent une contrainte supplémentaire sur le nombre de variables fixées à 1 dans une solution optimale du problème, ce qui leur permet de diviser l'espace des solutions réalisables en plusieurs parties. Ils déterminent une solution initiale associée à chaque partie de l'espace de recherche à l'aide d'une heuristique d'arrondi de la relaxation en continu, et applique un processus tabou avec une gestion dynamique de la liste tabou autour de chacun des points ainsi généré. Dans le second article, les auteurs intègrent un ordonnancement plus « intelligent » des différentes parties de l'espace des solutions réalisables pour essayer d'obtenir au plus vite une meilleure solution qui permet d'accélérer le reste de la recherche. Ils utilisent également une méthode pour fixer un sous-ensemble de variables. Cette méthode repose sur la définition de deux problèmes associés aux variables dites les *plus fractionnaires* (celles qui minimisent la différence avec 0.5) de la solution de la

relaxation en continu de chaque sous-espace exploré. Pour chacune de ces variables, les deux problèmes correspondant au cas où elles sont fixées à 0 ou 1 sont résolus. Une variable est alors fixée si elle est toujours égale à 0 ou 1 dans l'ensemble des solutions obtenues. Cette procédure de fixation permet d'exécuter l'algorithme de recherche tabou sur des problèmes nettement réduits et ainsi d'améliorer les résultats obtenus précédemment. Les résultats publiés dans ces deux articles servent aujourd'hui de référence sur un ensemble important d'instances existantes, même si un inconvénient majeur de ces deux approches résident dans la complexité temporelle très importante pour résoudre des problèmes de grandes tailles. Nous reviendrons sur cette complexité dans les chapitres suivants lors de la présentation de nos résultats numériques.

Terminons cette section en mentionnant que la recherche tabou est une métaheuristique convergente sous certaines conditions comme le montrent Hanafi [Han01] et Glover et Hanafi [GH02].

2.3.3 Le recuit simulé

Les premières références décrivant des métaheuristiques pour le problème *SADM* sont dues à Drexl [Dre88], Dueck et Scheuer [DS90] et décrivent des algorithmes de recuit simulé. Cette méthode est aussi abordée pour ce problème dans Hanafi, Fréville et El Abdellaoui [HFA96]. Le recuit simulé est inspiré d'un processus utilisé en métallurgie. Il alterne des cycles de refroidissement lent et de réchauffage (*recuit*) qui tendent à minimiser l'énergie du matériau. Dans le cas de la métaheuristique, la fonction à optimiser (minimiser ou maximiser) devient l'énergie du système. Un paramètre représentant la température du système est également nécessaire. Au cours des itérations de l'algorithme, la visite d'une solution améliorant la valeur de l'objectif permet de faire baisser la température du système. Cela tend à terme à conduire la recherche dans un optimum local. Il est possible d'accepter une solution dégradant l'état du système pour diversifier la recherche, et surtout éviter de rester bloqué dans un optimum local. Chaque modification de la solution courante modifie l'état du système et entraîne une variation de l'énergie de celui-ci. Cette variation est calculée à partir de la fonction à optimiser. Si elle est négative, elle est appliquée à la solution courante car elle fait baisser l'énergie du système. Dans le cas contraire, elle est acceptée selon une certaine probabilité. La gestion de la température est un élément important de l'algorithme. Deux cas de figures sont généralement rencontrés. Dans le premier cas, la

température ne change que lorsque le système a atteint un *équilibre thermodynamique*. Cela se traduit d'un point de vue algorithmique par un certain nombre de modifications élémentaires effectuées et correspond à définir des paliers de température. Dans le second cas, la température diminue de façon continue. Il faut alors définir une fonction de décroissance.

Les principaux inconvénients du recuit simulé résident dans le choix des nombreux paramètres tels que la température initiale, la loi de décroissance de la température, les critères d'arrêt ou la longueur des paliers de température. Ces paramètres sont souvent choisis de manière empirique. Il a également été montré que cette métaheuristique converge vers une solution optimale du problème sous certaines conditions.

2.3.4 Les algorithmes basés sur des populations

Différents algorithmes dits évolutionnaires peuvent aussi être référencés dans la littérature. Ils se basent sur la notion de population de solutions. Plusieurs types d'algorithmes existent comme les algorithmes génétiques ou les algorithmes de recherche dispersée par exemple. La seconde catégorie rentre dans le cadre de notre chapitre 4, c'est pourquoi nous ne nous y attardons pas ici. Il existe à notre connaissance peu de références sur cette approche pour le problème *SADM*. Les algorithmes génétiques sont beaucoup plus courants. Ils sont apparus en 1975 et ont été proposés par Holland [Hol75]. Leur principe est simple et repose sur la définition des trois points suivants :

- Un codage des solutions sous la forme de chaînes d'éléments insécables. Ces chaînes sont assimilées à des chromosomes.
- Des moyens pour reproduire en grand nombre ces chaînes.
- Une fonction d'adaptation permettant d'évaluer la qualité de chacune des chaînes créées au cours du processus.

Un algorithme génétique consiste à répéter les quatre opérations suivantes sur la population de solutions tant qu'un critère d'arrêt choisi n'est pas rencontré : évaluation de la population, sélection d'un ensemble d'individus, reproduction de ces individus et remplacement d'une partie de la population.

A une itération donnée, les solutions de la population sont donc évaluées (une note leur est attribuée). Une sélection est ensuite effectuée. Il existe différentes techniques de sélection telles que la sélection par rang (les meilleurs individus sont conservés), par tournoi (des paires d'individus sont choisies et ensuite le meilleur individu de chaque paire est

sélectionné), uniforme (sélection aléatoire et sans intervention de la capacité d'adaptation), *etc*.

Lorsque deux individus ont été sélectionnés, un opérateur de croisement (ou recombinaison ou encore crossing-over selon la terminologie anglaise) est appliqué pour générer un nouvel individu. L'exemple suivant illustre cette technique dans le cas d'un codage binaire des individus.

Exemple 2.2 : opérateur de croisement dans un algorithme génétique

parent 1	1	1	0	0	■	1	1	1	0	1	0
parent 2	1	0	0	1	■	0	1	1	1	0	0
fils	1	1	0	0	■	0	1	1	1	0	0

Dans cet exemple, le point de rupture (ou point de croisement) se situe après le quatrième élément parmi les dix qui représentent un individu. L'individu *fils* reçoit les quatre premiers éléments du *parent* 1, et les six derniers du *parent* 2.

Notons qu'il existe aussi des opérateurs de croisement multiples, c'est-à-dire reposant sur plusieurs points de croisement. Un autre opérateur génétique est souvent utilisé. Il consiste à effectuer des mutations sur une petite partie de la population. Une mutation consiste à modifier un gène (une variable dans l'exemple précédent) choisi aléatoirement au sein d'un chromosome (d'un individu). Ainsi dans le cas d'un codage binaire, un gène peut se transformer de 0 en 1 ou de 1 en 0. Il est nécessaire de définir un taux de mutation peu élevé de manière à ne pas transformer l'algorithme en une recherche aléatoire, et ainsi conserver les principes d'évaluation et de sélection. L'opérateur de mutation permet d'intégrer de la diversité au sein du processus en évitant que la population n'évolue plus ou très peu.

La phase de remplacement consiste à construire la nouvelle population soit en prenant les individus les plus performants sur l'ensemble des deux générations d'individus, soit en supprimant simplement une partie des éléments les moins performants selon un critère propre à l'application (généralement une taille maximale pour la population). Cette phase se base sur la théorie de Darwin au sujet de la sélection naturelle : les individus les plus adaptés tendent à survivre le plus longtemps et à se reproduire le plus aisément.

Le processus est réitéré un grand nombre de fois de manière à imiter le principe d'évolution, qui ne prend son sens que sur un nombre important de générations. Différents critères d'arrêt peuvent être utilisés comme un nombre arbitraire de générations ou lorsque

qu'une solution possède une évaluation suffisamment satisfaisante par exemple. Différentes techniques pour améliorer le fonctionnement de ces algorithmes peuvent être utilisées comme par exemple l'insertion d'individus non issus de la descendance de la génération précédente de manière à diversifier encore plus la recherche.

Les points faibles des algorithmes génétiques résident généralement dans le temps d'exécution assez important pour pouvoir converger vers une population d'individus élites, et le nombre de paramètres important à régler. Ce type de méthode repose également beaucoup sur l'aléatoire. La gestion de la diversité n'est pas toujours des plus aisée pour éviter les optima locaux, même si nous avons mentionné plusieurs moyens permettant d'assurer un minimum de diversité. Les premières références sur ce sujet pour le problème *SADM* n'ont pas donné de bons résultats pour ce problème. Thiel et Voss [TV94] ont montré qu'un algorithme génétique standard appliqué à l'ensemble de l'espace de recherche ne pouvait pas permettre l'obtention de solutions de bonne qualité pour le problème *SADM* en général. Chu et Beasley [CB98] ont proposé le premier algorithme génétique efficace pour ce problème. La recherche est limitée aux solutions réalisables. Citons aussi le travail de Haul et Voss [HV98] qui ont obtenu de bons résultats en introduisant des relaxations surrogates au sein d'un algorithme génétique.

D'autres métaheuristiques basées sur des populations existent. Sörensen et Sevaux [SS06] ont récemment proposé un algorithme appelé MA|PM pour *memetic algorithm and population management* selon la terminologie anglaise. Un algorithme *mémétic*, encore appelé algorithme génétique hybride, combine la recherche locale et des opérateurs de croisement issus des algorithmes génétiques. Les auteurs ajoutent plusieurs éléments permettant de contrôler la diversité d'une partie de la population. Ils définissent pour cela plusieurs distances entre les solutions ainsi que plusieurs critères de sélection des solutions à conserver dans la population. Quelques résultats obtenus pour le problème *SADM* sont mentionnés, mais uniquement sur des instances de petites tailles. L'apport du management de la population permet d'améliorer le comportement de l'algorithme génétique hybride utilisé seul.

Citons enfin le travail de Puchinger, Raidl et Gruber [PRG05] qui combine un algorithme mémétic et un algorithme de séparation et coupes pour le problème *SADM*. Les deux algorithmes sont utilisés de manière parallèle et échangent la meilleure solution lorsque celle-ci est améliorée. Les résultats obtenus sur un ensemble d'instances corrélées et difficiles

sont intéressants et les solutions obtenues sont de bonne qualité avec un temps d'exécution non excessif.

2.3.5 Autres heuristiques

Nous abordons pour terminer quelques autres heuristiques et métaheuristiques existantes. Une méthode assez connue est l'algorithme GRASP (*Greedy Randomized Adaptative Search Procedure* en anglais) (Feo et Resende [FR89]). Il s'agit d'une procédure itérative qui peut être vue comme une approche hybride. Elle combine en effet les principes des heuristiques gloutonnes, de la recherche aléatoire et des méthodes de voisinage. Un algorithme GRASP répète le schéma suivant composé de deux étapes : la construction d'une solution puis une phase de descente pour améliorer cette solution. A notre connaissance une des seules applications de cette approche au problème *SADM* est due à Chardaire, McKeown et Maki [CMM01].

La méthode de bruitage proposée par Charon et Hudry [CH93] consiste à répéter le processus suivant. *Bruiter* les données à partir de la solution courante en changeant les valeurs considérées dans l'objectif du problème. Une méthode de descente est ensuite appliquée en utilisant cette fonction bruitée. A chaque itération l'amplitude du bruitage diminue de manière à retrouver l'objectif initial à la fin du processus. Le bruitage des données fait appel à plusieurs éléments : une fonction aléatoire, la plus grande donnée du problème et un paramètre permettant de normaliser le niveau de bruit par rapport à l'ensemble des données. Cette approche a été introduite pour résoudre les problèmes comportant des données réelles et a par exemple été appliquée au problème du voyageur de commerce ou au problème *SADM* par Hanafi, Fréville et El Abdellaoui [HFA96].

La métaheuristique appelée colonie de fourmis est basée sur une simulation simplifiée du comportement réel des fourmis (Dorigo et Di Caro [DC99]) grâce à des fourmis artificielles se déplaçant dans un graphe représentant le problème à résoudre. Les fourmis cherchent de la nourriture et se déplacent dans le graphe de façon quasi aléatoire en laissant derrière elles une substance chimique appelée phéromone qui leur permet de faire le chemin retour vers leur nid une fois la nourriture trouvée. La phéromone a la propriété de s'évaporer avec le temps. Le fait de repasser plusieurs fois par le même endroit permet aux fourmis de renforcer la présence de phéromone et de générer un plus court chemin vers la nourriture. Fidanova [Fid05] ou encore Alaya, Solnon et Ghedira [ASG04] ont appliqué ce genre d'algorithme au problème *SADM* dans lequel ils testent plusieurs heuristiques pour définir le

mouvement des fourmis. Ces mouvements reposent sur une probabilité pour chaque fourmi d'aller d'un état à un autre, chaque état représentant une solution partielle du problème. Les résultats obtenus sur des instances existantes permettent de générer des solutions proches de celles obtenues par Chu et Beasley avec leur algorithme génétique [CB98].

Une autre métaheuristique appelée *Meta-RaPs* (pour *Metaheuristic with Randomized Priority search* en anglais) a aussi été récemment appliquée au problème *SADM* par Moraga, DePuy et Whitehouse [MDW05]. *Meta-RaPs* est une stratégie de haut niveau utilisée pour construire et améliorer des solutions réalisables du problème à l'aide de règles heuristiques simples basées sur l'aléatoire. L'aléatoire est utilisé pour éviter à la recherche de rester bloquée dans un optimum local. Dans cet article *Meta-RaPs* est appliqué avec quatre heuristiques gloutonnes. L'utilisation de cette méthode permet d'améliorer le comportement des heuristiques gloutonnes seules sur des problèmes de petites tailles. Les résultats obtenus sur des instances de plus grandes tailles sont meilleurs que ceux de Haul et Voss [HV98] mais restent un peu inférieurs à ceux de Chu et Beasley [CB98]. Le temps d'exécution de cette approche est cependant assez faible et le rapport entre la qualité de la solution et la complexité temporelle apparaît donc intéressant.

Plusieurs approches récentes font aussi appel à l'utilisation de structures de voisinage particulières. Ces voisinages dépendent de relaxations et évoluent au cours du processus.

Shaw décrit dans [Sha98] une méthode appelée *Recherche dans un voisinage large* pour résoudre un problème de tournées de véhicules. Cette méthode combine l'exploration de voisinages comme dans les méthodes de recherche locale et l'utilisation d'un arbre de propagation de contraintes pour évaluer le coût et la légitimité d'un mouvement comme dans les méthodes de programmation par contraintes. L'évaluation des mouvements possibles devient plus longue mais le nombre de candidats possibles est plus petit.

Fischetti et Lodi [FL03] ont développé un autre type d'approche pour résoudre les problèmes en nombres entiers mixtes, le branchement local. Le but de la méthode est d'explorer différents voisinages de petites tailles en utilisant un algorithme exact de séparation et évaluations (ou séparation et coupes) comme boîte noire. A chaque nœud de l'arbre de recherche est introduit une coupe pour définir le voisinage à explorer. Cette coupe est générée à partir de la solution courante du problème et en fonction d'un paramètre pour contrôler la taille du voisinage. Elle permet de définir un voisinage de type *k*-opt (en référence au problème de voyageur de commerce) et de partitionner l'espace de recherche associé au nœud courant en deux sous-espaces. Le but est d'obtenir une meilleure solution que la solution courante le plus rapidement possible en explorant un sous-espace de recherche le

plus petit possible. Le processus est répété en utilisant la nouvelle meilleure solution comme point de départ et s'arrête lorsqu'il n'est plus possible d'améliorer la solution courante. Les auteurs ont aussi intégré deux processus de diversification pour améliorer les performances de l'algorithme. Ils sont utilisés lorsque la solution courante n'a pas été améliorée. Les résultats obtenus sur un ensemble d'instances existantes de problèmes en nombres entiers mixtes montrent l'efficacité de l'approche qui permet généralement d'améliorer les résultats du logiciel d'optimisation CPLEX de Ilog dans une version récente (7.0). Lichtenberger [Lic05] a appliqué cette approche au problème du *SADM* dans le cadre de ces travaux de recherche de doctorat. Les résultats obtenus sur différents ensembles d'instances disponibles sur Internet montrent que l'intégration de la méthode de branchement local dans le logiciel libre COIN/BCP [RL01] permet une amélioration importante de ses performances. Les expériences numériques ont surtout été menées dans le but de montrer que l'approche permet de diminuer sensiblement le nombre de nœuds de l'arbre de recherche et donc d'accélérer le processus. Les solutions obtenues sont de bonne qualité en des temps raisonnables et finalement les améliorations sont plus sensibles lorsque la taille des instances augmente.

Dans le même cadre que Fischetti et Lodi, Danna, Rothberg et Pape [DRP05] ont proposé une méthode d'exploration de voisinages induits par des relaxations pour résoudre les problèmes en nombres entiers mixtes. L'approche consiste à générer un voisinage prometteur à partir de la relaxation en continu au cours d'un algorithme de séparation et coupes à certains nœuds de l'arbre de recherche. Un sous-problème à résoudre de manière optimale est défini en fonction du voisinage de la solution courante à chaque itération. Plusieurs concepts différencient cette approche du branchement local. Le voisinage est défini à partir de la solution courante et de la solution de la relaxation en continu, ce qui permet d'apporter plus d'informations lorsque la solution courante est de faible qualité. L'algorithme peut être appelé à n'importe quel nœud de l'arbre de recherche alors que le branchement local n'intervient que si une amélioration de la solution est obtenue. Enfin les sous-problèmes résolus exactement sont plus petits et sont donc résolus plus rapidement.

Citons pour terminer l'article de Hanafi, Fréville et El Abdellaoui [HFA96] qui présente de nombreuses approches heuristiques existantes sur le problème et donne quelques résultats numériques, ainsi que certains pseudo codes d'heuristiques facilement réutilisables. Il existe aussi plusieurs articles qui référencent de nombreuses annotations bibliographiques sur le problème *SADM* comme ceux de Chu et Beasley [CB98], de Fréville [Fré04], ou encore de Fréville et Hanafi [FH05].

2.4 Conclusions

Ce chapitre nous a permis de passer en revue un ensemble de méthodes de résolution existantes pour les problèmes en nombres entiers et en nombres entiers mixtes. Nous avons plus particulièrement insisté sur les algorithmes proposés pour résoudre le problème du sac-à-dos multidimensionnel. Il ressort de cet état de l'art que les méthodes exactes semblent réellement limitées pour résoudre ce problème. Plusieurs méthodes permettent de déterminer des bornes supérieures efficaces à l'aide de relaxations surrogates par exemple, mais cela ne permet toutefois pas de définir un processus de résolution exacte aussi performant que certaines heuristiques ou métaheuristiques.

Devant ce constat, nous nous sommes focalisés sur des méthodes approchées permettant d'obtenir des solutions réalisables le plus proche possible de l'optimum. Les métaheuristiques représentent les algorithmes généralement les plus efficaces pour résoudre les problèmes NP-Difficiles. Nous retenons également la méthode de branchement local qui semble pouvoir singulièrement améliorer le comportement d'un algorithme de séparation et coupes. Dans l'ensemble des métaheuristiques proposées pour résoudre le problème *SADM* et les problèmes en variables 0-1, les algorithmes de recherche tabou ont prouvé depuis quelques années leur efficacité pour obtenir de très bons résultats. Différentes méthodes hybrides très efficaces ont aussi été recensées et il nous semble que ce genre d'approches soit le plus à même de nos jours de permettre la définition de processus de résolution efficaces. Les trois chapitres suivants de ce mémoire présentent des algorithmes basés sur plusieurs méthodes de résolution pour les problèmes en variables 0-1. Nous présentons dans le chapitre suivant un algorithme utilisant la programmation dynamique et la recherche tabou au sein d'un processus dit d'intensification globale. Le chapitre quatre aborde un algorithme de recherche dispersée qui intègre plusieurs composants de la recherche tabou et de la méthode des chemins reliants pour intensifier la recherche. Finalement le chapitre cinq décrit plusieurs heuristiques basées sur l'utilisation de relaxations pour générer des bornes inférieures et supérieures du problème de bonne qualité.

Ces trois chapitres sont présentés de manière à ce qu'ils soient lisibles le plus indépendamment possible du reste de ce mémoire. Certaines références entre les chapitres sont cependant faites lorsque cela nous paraît réellement utile pour ne pas surcharger la présentation.

Chapitre 3

Recherche tabou : intensification globale avec la programmation dynamique

Nous présentons dans ce chapitre un algorithme basé sur la coopération entre une méthode exacte, la programmation dynamique et une méthode approchée, la recherche tabou, pour la résolution de problèmes en variables 0-1. L'approche proposée est applicable à différents problèmes d'optimisation combinatoire, et plus précisément à ceux pour lesquels il est possible de définir un algorithme de programmation dynamique efficace. L'idée originale de l'approche est de décomposer l'ensemble des variables de décision du problème en deux sous-ensembles. Après avoir déterminé une solution initiale du problème à l'aide d'une heuristique adaptée, nous appliquons la programmation dynamique avec des propriétés de réduction sur une petite partie de l'ensemble des variables correspondant au premier sous-ensemble et générons une liste comportant les solutions optimales des problèmes réduits associés. La recherche tabou est ensuite appliquée sur le sous-ensemble de variables complémentaire dans le but d'améliorer la solution courante. La coopération entre les deux approches se situe au niveau de l'exploration du voisinage dans l'algorithme tabou. Chaque mouvement nécessite en effet l'accès à la liste générée par la programmation dynamique pour obtenir une solution sur l'ensemble des variables du problème original. Le processus peut ainsi être vu comme une intensification globale car il existe deux niveaux d'intensification : le premier correspond à l'intensification classique telle qu'elle est définie dans la recherche tabou, et le second correspond à l'utilisation de la liste de programmation dynamique lors de l'évaluation des voisins de la solution courante pendant la recherche tabou.

L'approche a été développée et testée sur le problème du sac-à-dos multidimensionnel. Les résultats obtenus sont encourageants et montrent que l'algorithme est performant lorsque le nombre de variables devient conséquent et que le nombre de contraintes reste peu élevé (Wilbaut et al. [WHFB06]). Ce chapitre est organisé selon le plan suivant. Nous commençons par présenter les composants de l'algorithme de programmation dynamique avec différents principes de réduction. Nous abordons ensuite la recherche tabou et le processus

d'intensification globale. Nous illustrons alors le déroulement de l'algorithme sur un problème de petite taille. Nous donnons enfin les résultats numériques, avant de terminer par les conclusions de ce travail.

3.1 Programmation dynamique avec réduction

La programmation dynamique (*PD*) est une méthode exacte formulée par Bellman à la fin des années 1950 [Bel57]. Elle est basée sur le principe d'optimalité, et peut être appliquée à de nombreux problèmes complètement différents tels que des problèmes statistiques, continus, discrets, *etc*. Elle est utilisable lorsque le problème à résoudre est décomposable en étapes. On parle plus précisément de décisions en étapes.

Cette approche est une des méthodes exactes les plus importantes dans la littérature pour la résolution des problèmes d'optimisation. Différentes applications efficaces ont été proposées pour le problème du sac-à-dos (Toth [Tot80]). Les algorithmes basés sur la programmation dynamique sont généralement faciles à implémenter et très efficaces pour les problèmes de petites et moyennes tailles. La programmation dynamique ne s'adresse cependant pas à tous les types de problèmes d'optimisation combinatoire et son application sur des instances de grandes tailles est généralement très coûteuse voire impossible pour des raisons de complexité spatiale et/ou temporelle. Cela se vérifie en particulier pour les problèmes de la classe NP-difficile.

La programmation dynamique permet généralement de définir une méthode efficace pour les problèmes de grandes tailles lorsqu'elle est hybridée avec une autre approche. La situation n'est pas toujours aussi facile avec le problème du sac-à-dos multidimensionnel (*SADM*). Lorsque le nombre de contraintes augmente, la quantité d'informations nécessaires à conserver en mémoire devient rapidement très importante. Notre approche repose en partie sur l'algorithme de programmation dynamique initialement proposé par Balev et al. [BYFA01] au problème *SADM* qui généralise les travaux présentés par Viader [Via98] pour le problème *SAD*. Pour clarifier la présentation nous nous plaçons dans la suite de ce chapitre dans le cas du problème *SADM*. Les relations de récurrence pour ce problème sont obtenues à partir du problème noté $SADM(k, g)$, défini comme suit.

$$SADM(k,g) \quad \begin{cases} \max & \sum_{j=1}^{k} c_j x_j \\ s.c. & \sum_{j=1}^{k} a_{ij} x_j \le g_i \quad \forall i = 1,...,m \\ & x_j \in \{0,1\} \quad \forall j = 1,...,k \end{cases}$$

avec k entier $\in [0,n]$ et g entier tel que $0 \le g \le b$.

En notant $f(k,g)$ la valeur optimale du problème $SADM(k,g)$, nous pouvons alors définir les relations de récurrence suivantes.

$$f(k,g) = \begin{cases} f(k-1,g) & \text{si } a_k \not\le g \\ \max\{f(k-1,g); c_k + f(k-1,g-a_k)\} & \text{si } a_k \le g \end{cases} \qquad (3.1)$$

pour $k = 1,2,\ldots,n$ avec les conditions initiales $f(0,g) = 0$, pour $0 \le g \le b$. Notons que $f(n,b)$ correspond à la valeur optimale du problème *SADM*.

La complexité spatiale de cette approche dans le pire des cas est en $O(n \times b_1 \times b_2 \times \ldots \times b_m)$. De manière à limiter les ressources mémoires nécessaires à l'application de cette approche, nous avons utilisé un algorithme de programmation dynamique creux, basé sur une liste de programmation dynamique et sur le fait que la fonction $f(k,g)$ est croissante par morceaux. L'idée principale est de maintenir à jour une liste L d'états. Chaque état est composé de $(m+2)$ éléments. A une itération donnée k, un élément de la liste comporte une valeur associée à l'objectif du problème notée v, un ensemble de m capacités restantes pour les $n - k$ variables, noté β, et le numéro de l'itération (l'étape), k. Ce dernier composant est utile pour déterminer une solution optimale lors de la phase de retour arrière de la programmation dynamique. L'algorithme se décompose en deux phases : une première phase consiste à construire la liste L, et une seconde phase à reconstruire une solution optimale du problème. Dans la suite de ce chapitre, nous référençons les éléments de la liste par (v,β,k) ou simplement par (v,β) lorsque la précision de k n'est pas nécessaire à une meilleure compréhension.

L'algorithme 3.1 présente le fonctionnement de l'algorithme de construction de la liste de programmation dynamique.

Algorithme 3.1 Construction de la liste de programmation dynamique pour le *SADM*

Entrée : le problème *SADM*.
Sortie : la liste générée L.

```
1 : L = (0,b);
2 : pour k = 1 à n faire
3 :     L1 = ∅;
4 :     pour chaque état (v,β,k) dans L faire
5 :         si β ≥ a_k alors L1 = L1 ∪ {v + c_k, β – a_k , k);
6 :     fin pour
7 :     L = L ∪ L1;
8 : fin pour
```

Différentes améliorations permettent de diminuer la complexité spatiale d'un algorithme de programmation dynamique. L'utilisation de techniques de dominance en est un exemple. Soient $s_1 = (v_1, \beta_1)$ et $s_2 = (v_2, \beta_2)$ deux éléments contenus dans la liste de programmation dynamique, l'élément s_1 domine l'élément s_2 si $v_1 \geq v_2$ et $\beta_1 \geq \beta_2$. Dans cette situation, l'élément s_2 peut clairement être éliminé de la liste puisqu'il ne pourra pas produire de solutions meilleures que celles obtenues à partir de l'élément s_1. L'élimination des éléments dominés au cours du processus de programmation dynamique a surtout été utilisée dans le cas du problème du sac-à-dos. Cependant lorsque m augmente, la détection des éléments dominés peut devenir très coûteuse. Nous avons mesuré le nombre d'éléments dominés dans la liste de programmation dynamique pour évaluer l'intérêt d'intégrer ce genre de méthodes pour le problème *SADM*. Il s'avère que sur un ensemble de 270 instances décrites dans la section 3.4 le pourcentage d'éléments dominés dans la liste lorsque la programmation dynamique est appliquée sur un petit nombre de variables comme c'est le cas ici est très faible. Il varie entre 0% et 3%, avec une moyenne de 0.04%. Ce résultat montre qu'il n'y a pas de réel intérêt à mettre en œuvre une méthode coûteuse dont l'exécution ne sera pas compensée par le gain en place mémoire dans la liste.

Malgré l'utilisation d'un algorithme de programmation dynamique creux, cette approche reste inadaptée pour la résolution d'instances de problèmes *SADM* de grandes tailles en raison des complexités spatiales et temporelles. Les premières étapes de l'algorithme sont cependant très rapides et permettent d'obtenir des informations utilisables pour fixer des variables à leurs valeurs optimales comme nous allons le voir dans la suite. Nous appliquons ainsi l'algorithme de programmation dynamique sur une partie des variables uniquement. Le nombre de variables concerné, noté *n'* dans la suite, est déterminé de manière expérimentale en fonction des capacités de la machine et de la taille de l'instance (en particulier le nombre

de contraintes). Une estimation de *n'* peut toutefois être faite comme nous le montrons dans la section 3.4. Le choix des variables considérées dans l'algorithme de programmation dynamique est guidé selon des propriétés de réduction.

Différentes techniques de prétraitement existent pour améliorer la résolution de programmes en variables entières telles que les méthodes de fixation de variables et d'élimination de contraintes redondantes mentionnées par exemple dans l'article de Savelsbergh [Sav94]. Dans notre approche nous utilisons uniquement des règles de réduction permettant de fixer certaines variables à leur valeur optimale.

Les principes de réduction dépendent de deux séquences de bornes inférieures $\{l_j\}_{j\in N}$ et supérieures $\{u_j\}_{j\in N}$ associées aux variables du problème et déterminées à partir d'une solution réalisable du problème x^0. Nous utilisons également le terme de problème réduit. Un problème réduit est défini à partir d'une solution réalisable du problème x^0 et d'un sous-ensemble de variables $J \subseteq N$.

$$SADM(x^0,J) \quad \begin{cases} \max & cx \\ s.c. & Ax \le b \\ & x_j = x_j^0 \quad \forall j \in J \\ & x_j \in \{0,1\} \quad \forall j \in N \end{cases}$$

Dans la suite nous référençons par $SADM(x^0,\{j\})$ le problème dans lequel seule la variable x_j est fixée à sa valeur dans la solution x^0. De même, nous notons $SADM(e\text{-}x^0,\{j\})$ le problème obtenu en fixant la variable x_j à $1\text{-}x_j^0$ (e est le vecteur de dimension n ayant toutes ses composantes égales à 1). Soit u_j, pour $j \in N$, une borne supérieure du problème $SADM(e\text{-}x^0,\{j\})$. Tout au long de ce chapitre nous supposons que les variables x_j sont triées selon l'ordre décroissant des bornes supérieures u_j.

$$u_1 \ge u_2 \ge \ldots \ge u_n \tag{3.2}$$

La propriété classique suivante permet dans certains cas de fixer la variable x_j à sa valeur optimale à partir d'une borne inférieure et du problème réduit associé $SADM(\text{e-}x^0,\{j\})$.

Propriété 3.1 :

Pour tout $j \in N$ et pour toute solution réalisable $x^0 \in \{0,1\}^n$ du problème *SADM*, si

$$v(SADM(e\text{-}x^0,\{j\})) \le cx^0$$

alors soit $x_j = x^0_j$ dans toute solution optimale du problème, soit x^0 est une solution optimale du problème.

Preuve :

Supposons qu'il existe un indice j dans N pour lequel $v(SADM(e\text{-}x^0,\{j\})) \leq cx^0$, et supposons que $x_j \neq x^0_j$ dans toute solution optimale du problème.
Comme x^0 n'est pas une solution optimale du problème, on a

$\exists x^*$ solution réalisable de *SADM* telle que $cx^* > cx^0$ et $x^*_j = 1 - x^0_j$.

Comme $v(SADM(e\text{-}x^0,\{j\})) \leq cx^0$, on a $cx^0 \geq cx^*$. D'où la contradiction. □

Nous utilisons dans la suite une autre approche basée sur la programmation dynamique et sur la relation entre les bornes inférieures et supérieures pour fixer des variables. Nous rappelons les propriétés principales sur lesquelles repose l'algorithme de programmation dynamique (Balev et al. [BYFA01]). La proposition suivante permet de déterminer si une borne inférieure obtenue à une itération donnée est une valeur optimale du problème.

Proposition 3.1 :

Soit x^0 une solution réalisable du problème *SADM* et soit

$$l_j = v\left(SADM\left(x^0,\{j+1,...,n\}\right)\right) \quad \forall j \in N.$$

Si $l_k \geq u_{k+1}$ pour un certain $k \in \{1,2,\ldots,n\text{-}1\}$ alors $v(SADM) = l_k$.

Preuve :

Supposons que $l_k \geq u_{k+1}$ pour un certain $k \in \{1,2,\ldots,n\text{-}1\}$ et soit x^* une solution optimale du problème *SADM*. Deux cas sont possibles.
Premier cas : si $\forall j \in \{k+1,...,n\}\ x^*_j = x^0_j$, alors la valeur optimale du problème *SADM* est clairement égale à l_k par définition de l_k elle-même (la valeur optimale du problème dans lequel toutes les variables comprises entre k+1 et n sont fixées à leur valeur dans la solution x^0).
Deuxième cas : si $\exists j \in \{k+1,...,n\}\ x^*_j \neq x^0_j$, alors on a $u_{k+1} \geq u_j$ (car $j \geq k$+1 et d'après 3.2). Par définition de u_j on a aussi $u_j \geq v(SADM(e\text{-}x^0,\{j\})$. De plus $v(SADM(e\text{-}x^0,\{j\}) = v(SADM)$ (car $x^*_j = 1 - x^0_j$). D'où $l_k \geq u_{k+1} \geq v(SADM)$. Comme par définition $v(SADM) \geq l_k$ on a bien $l_k = v(SADM)$. □

Les valeurs des bornes supérieures $\{u_j\}_{j\in N}$ sont déterminées dans nos expériences numériques à l'aide de la relaxation en continu. L'utilisation de bornes plus raffinées est envisageable (Fréville et Plateau [FP94]). Notons que d'après la définition des bornes inférieures $\{l_j\}_{j\in N}$ on a

$$l_1 \le l_2 \le \ldots \le l_n \tag{3.3}$$

En effet $l_j \ge l_{j-1}$ pour tout j dans $\{2,\ldots,n\}$ puisque dans le problème associé à la borne l_j il y a une variable de moins qui est fixée que dans le problème associé à l_{j-1}. Nous expliquons dans la proposition suivante le mécanisme permettant de fixer des variables à leur valeur optimale.

Proposition 3.2 :

Supposons qu'il existe une paire d'indices (h,k) tels que $h < k < n$ et $l_h \ge u_k$. Dans ces conditions il existe une solution optimale x^* du problème *SADM* telle que

$$x_j^* = x_j^0 \quad \forall j \in \{k,\ldots,n\}.$$

Preuve :

Supposons que les hypothèses de la proposition soient vérifiées. On a :

$h < k$ => $h \le k - 1$

=> $l_h \le l_{k-1}$ d'après (3.3)

=> $u_k \le l_h \le l_{k-1}$ par hypothèse

=> $v(SADM) = l_{k-1}$ par application de la proposition 3.1.

=> $\forall j = k,\ldots,n \quad x_j^* = x_j^0$ par définition de $l_{k-1} = v(SADM(x^0,\{k,\ldots,n\}))$. □

Les propositions précédentes sont intégrées dans l'algorithme de programmation dynamique qui est exécuté sur le sous-ensemble $N' = \{1,2,\ldots,n'\}$ avec $n' \le n$. La programmation dynamique est ainsi à la fois utilisée comme une heuristique (car uniquement sur un sous-ensemble de variables), mais aussi comme une méthode exacte pour résoudre de manière optimale un sous-espace de recherche de petite taille par rapport à $|N|$. L'algorithme peut aussi éventuellement prouver l'optimalité de la solution courante.

La séquence de bornes inférieures $\{l_j\}_{j\in N}$ est déterminée au cours de l'algorithme de programmation dynamique. A une itération k donnée, l_k est calculée comme suit :

$$l_k = \sum_{j=k+1}^{n} c_j x_j^0 + \max\left\{ v \,\middle|\, \beta \geq \sum_{j=k+1}^{n} A^j x_j^0, (v,\beta) \in L \right\} \quad (3.4)$$

Lemme 3.3 :

La valeur de la borne inférieure l_k définie par (3.4) est égale à $v(SADM(x^0,\{k+1,\ldots,n\}))$.

Preuve :

Chaque élément (v,β) (pour les k premières itérations) de la liste L correspond à une solution réalisable x par définition de L, et vérifie : $v = \sum_{j=1}^{k} c_j x_j$ et $\beta = b - \sum_{j=1}^{k} A^j x_j$. Le fait d'avoir $\beta \geq \sum_{j=k+1}^{n} A^j x_j^0$ dans (3.4) assure que la solution associée au second terme de (3.4) soit réalisable. De plus, le fait de prendre le maximum assure que la valeur de l_k est optimale par rapport à la solution partielle x^0 sur l'ensemble $\{k+1,\ldots,n\}$. □

L'algorithme 3.2 décrit l'algorithme de programmation dynamique obtenu. Nous introduisons une variable x_k à chaque étape de l'algorithme dans la phase de construction. Nous calculons ensuite une borne inférieure en complétant le meilleur élément dans la liste L par $(x_{k+1}^0,....,x_n^0)$. Nous pouvons ensuite essayer de fixer une ou plusieurs variables par application des propriétés précédentes.

Ajoutons quelques remarques complémentaires sur cet algorithme. Tout d'abord, il n'est pas nécessaire de calculer les bornes inférieures à chaque itération. En effet sur les problèmes de grandes tailles, il est peu probable de réussir à fixer toutes les variables en un nombre peu élevé d'itérations. C'est pourquoi nous ne calculons en fait que la borne inférieure correspondant à la dernière variable traitée par la programmation dynamique, ce qui permet d'accélérer le processus sur les premières itérations. Remarquons aussi que le processus est répété sur le problème réduit tant que des nouvelles variables peuvent être fixées.

Algorithme 3.2 Programmation dynamique creuse avec réduction pour le problème *SADM*

Entrée : le problème *SADM*.
Sortie : une borne inférieure du problème $l_{n'}$.

```
1 : Soit x^0 une solution réalisable pour SADM;
2 : Calculer la séquence de bornes supérieures {u_j}_{j∈N} et trier les variables selon (3.2);
3 : Calculer n' et soit N' = {1,2,…,n'};
4 : L = {(0,b,0)}; k = 1;
5 : tant que k ≤ n' faire
6 :    L_1 = ∅;
7 :    pour chaque état (v, β) dans L faire
8 :       si β ≥ a_k alors L_1 = L_1 ∪ {v + c_k, β – a_k , k};
9 :    fin pour
10 :   L = L ∪ L_1;
11 : fin tant que
12 : Calculer l_{n'} selon (3.4)
13 : si l_{n'} ≥ u_{n'+1} alors stop (l_{n'} est la valeur optimale du problème par la proposition 3.1)
14 : sinon
15 :       Soit k un indice tel que u_{k-1} > l_{n'} ≥ u_k
16 :       si k existe alors
17 :          Ajouter les contraintes x_j = x_j^0 pour j = k,…,n au problème SADM (proposition 3.2)
18 :          Répéter le processus sur le problème obtenu
19 :       sinon stop (plus de fixation possible)
20 :       fin si
21 : fin si
```

Un exemple d'algorithme récent permettant d'obtenir des bornes inférieures de bonne qualité et utilisant la programmation dynamique de manière heuristique est dû à Bertsimas et Demir [BD02]. L'idée consiste à utiliser des bornes inférieures et supérieures pour approximer la valeur optimale des sous-problèmes *SADM*(k,g) définis précédemment, plutôt que de résoudre ces problèmes de manière optimale. En notant $L(k,g)$ (resp. $U(k,g)$) une borne inférieure (resp. supérieure) de la fonction $f(k,g)$, l'algorithme génère une solution approchée en commençant par déterminer la valeur de la variable x_n. Celle-ci est fixée à 1 si l'inégalité suivante

$$c_n + \varepsilon\, U\left(n-1, b-A^n\right) > \varepsilon\, U\left(n-1, b\right) \tag{3.5}$$

est respectée, avec ε qui représente le pourcentage de déviation maximum entre les bornes inférieures et supérieures

$$\varepsilon = \max\left\{\frac{L\left(n-1,b\right)}{U\left(n-1,b\right)}, \frac{L\left(n-1,b-A^n\right)}{U\left(n-1,b-A^n\right)}\right\} \tag{3.6}$$

La valeur de ε définie par (3.6) permet d'assurer que les valeurs optimales sont approximées de manière à ce que le pourcentage de déviation soit le même pour les deux

bornes supérieures utilisées. Le même principe est appliqué à la variable x_{n-1}, puis itérativement jusqu'à ce que toutes les variables aient été fixées ou qu'un critère d'arrêt soit rencontré.

L'efficacité de l'approche repose en partie sur le choix des méthodes de calcul des bornes inférieures et supérieures. Il y a en effet $2n$ calculs à faire. Les auteurs déterminent les bornes supérieures en tronquant les solutions de la relaxation en continu du problème associé (voir chapitre 2 section 2.2.2). Le calcul de la borne inférieure est effectué à l'aide d'une heuristique basée sur la programmation linéaire dont le principe est le suivant.

Après avoir généré une solution optimale $\bar{x}$ de la relaxation en continu du problème, les auteurs fixent l'ensemble des variables entières de cette solution. Les variables fractionnaires vérifiant $\bar{x}_j < \gamma$, avec γ paramètre (les auteurs mentionnent la valeur 0.25 comme une valeur permettant de faire la balance entre qualité de la solution et temps d'exécution) sont aussi fixées à 0. Le processus est répété sur le problème réduit ne comportant que les variables non fixées tant qu'il reste au moins une variable libre (en s'assurant qu'au moins une variable est fixée à chaque itération).

Les auteurs utilisent une procédure de fixation permettant d'accélérer le processus. Ils définissent aussi un critère d'arrêt qui permet de vérifier si une solution optimale est rencontrée au cours de la résolution. L'avantage de cette approche est qu'elle obtient des solutions réalisables d'assez bonne qualité (inférieure cependant à celles obtenues par Chu et Beasley [CB98] sur les instances existantes) et surtout en des temps d'exécution très faibles. L'algorithme permet également de rivaliser avec CPLEX dans sa version 6 sur des instances de grandes tailles générées aléatoirement.

Boyer, Elkihel et El Baz [BEB06] ont proposé un algorithme basé sur la programmation dynamique et la relaxation surrogate pour générer une borne inférieure du problème *SADM*. La relaxation surrogate est utilisée pour ramener le problème à un problème de type *SAD*. La programmation dynamique est alors appliquée sur ce problème et seules les solutions réalisables pour le problème *SADM* sont conservées dans une liste. Ils utilisent également des propriétés de réduction pour fixer des variables à leur valeur optimale, ainsi que des règles d'élimination d'états dominés au cours de la programmation dynamique. Ils proposent aussi une méthode d'amélioration de la borne inférieure générée par la programmation dynamique. Ils génèrent pour cela un problème réduit associé à chaque état de la liste. Le nombre de variables de chaque problème réduit dépend de l'étape associée à l'état.

Le problème obtenu est résolu à l'aide d'une méthode d'énumération implicite si sa taille n'est pas trop importante (en fonction d'un paramètre), ou alors à l'aide d'une heuristique gloutonne. Les résultats obtenus sur des instances existantes de petites tailles et sur des instances générées aléatoirement sont assez bons comparés à d'autres approches existantes.

3.2 Recherche tabou

Comme nous l'avons vu dans le chapitre 2, la recherche tabou (*RT*) est une méthode généralement efficace pour la résolution des problèmes d'optimisation difficiles. L'approche que nous proposons consiste à appliquer cette puissante métaheuristique en utilisant les informations générées par la programmation dynamique.

3.2.1 Intensification globale

Après la phase de programmation dynamique, le sous-espace associé au sous-ensemble de variables restantes est exploré à l'aide de la recherche tabou, et chaque solution partielle est complétée à l'aide des informations stockées durant la phase de programmation dynamique dans la liste.

Différentes analogies entre les algorithmes basés sur la résolution d'un problème *noyau* pour le problème du sac-à-dos et notre approche existent. Le but de ces algorithmes est d'isoler un sous-problème comportant les variables les plus *difficiles* à fixer. Balas et Zemel [BZ80] ont proposé la définition suivante pour ce problème. En supposant que les variables sont triées selon l'ordre décroissant des rapports c_j/a_j dits d'efficacité, et que x^* est une solution optimale du problème, le *noyau* est l'ensemble $No = \{p,\ldots,q\}$, avec $p = \min \{ j \in N \mid x_j^*=0 \}$ et $q = \max \{ j \in N \mid x_j^*=1 \}$. Les indices p et q ne sont bien sûr généralement pas connus à priori. La résolution du problème consiste dans le cas contraire à fixer l'ensemble des variables de 1 à p-1 à 1 et de q+1 à n à 0, puis à résoudre optimalement le problème noyau restant. Pisinger [Pis99b] montre que l'influence du problème noyau sur la difficulté des instances des problèmes de sac-à-dos générées aléatoirement est importante, en particulier pour les problèmes ayant un grand nombre de variables qui ont la même valeur d'efficacité. Les algorithmes basés sur le problème noyau pour résoudre le problème *SAD* consistent dans une première phase à générer le noyau (souvent de manière approximative), puis à le résoudre.

Nous pouvons voir notre approche comme une approche voisine de ces méthodes mais pour le problème *SADM*. Les variables traitées par la programmation dynamique forment ce qui peut être assimilé au problème noyau. Les problèmes réduits associés à ces variables sont ici résolus de manière exacte, ce qui permet de compléter les solutions partielles de la recherche tabou.

D'autres travaux avec le même état d'esprit peuvent être référencés. Ainsi, Pedroso [Ped04] a récemment développé un algorithme tabou pour la résolution des problèmes en nombres entiers mixtes utilisant une décomposition de l'ensemble des variables en deux sous-ensembles. Il propose de résoudre le problème associé aux variables entières par la recherche tabou, alors que les variables continues sont fixées par un algorithme du simplexe. Un mouvement effectué au cours de la recherche tabou consiste à incrémenter ou décrémenter d'une unité la valeur d'une variable entière du problème. Une relaxation en continu est ensuite résolue pour compléter la solution partielle obtenue.

Dans la suite de ce chapitre nous supposons que l'ensemble des indices des variables N est trié selon (3.2), et que N est partitionné de la manière suivante : $N = N' \cup N''$, $N' \cap N'' = \varnothing$ et $N' = \{1,2,...,n'\}$, $N'' = N - N' = \{n'+1, n'+2,...,n\}$ avec $n' = |N'|$ et $n'' = |N''|$. De la même manière une solution x et les données c et A du problème original peuvent être partitionnées avec les dimensions appropriées comme suit.

$$x = (x', x''),\ A = (A', A'') \text{ et } c = (c', c'') \text{ avec } x' = (x_j)_{\{j \in N'\}} \text{ et } x'' = (x_j)_{\{j \in N''\}}.$$

Le sous-ensemble N' correspond aux variables définissant la famille de sous-problèmes résolus de manière optimale, et N'' est le sous-ensemble correspondant aux variables sur lesquelles la recherche tabou influe.

Le voisinage de la solution courante est défini uniquement sur le sous-espace de recherche $\{0,1\}^{n''}$. Pour une solution x'' sur $\{0,1\}^{n''}$, l'ensemble des voisins est défini comme suit en modifiant une seule composante

$$V(x'') = \left\{ y \in \{0,1\}^{n''} \,\middle|\, Ay \le b \text{ et } \sum_{j \in N''} \left| x''_j - y_j \right| = 1 \right\} \tag{3.7}$$

Ce voisinage a été abordé dans le chapitre 2 (section 2.3.2). Son principal avantage est sa simplicité. La taille de ce voisinage est égale à $|N''|$. Son choix est guidé par le processus mis en place pour obtenir une solution sur N.

La valeur d'une solution z sur l'ensemble N générée à partir d'un voisin y d'une solution x'' dans $V(x'')$ est alors obtenue par le calcul suivant

$$cz = \sum_{j \in N''} c_j y_j + cx'^* \tag{3.8}$$

avec x'^* une solution optimale sur le sous-ensemble N' une fois que les variables de N'' sont fixées à leur valeur dans la solution y.

L'intégration de la programmation dynamique dans la recherche tabou se justifie par l'exploration de la liste L générée durant la phase de construction de la programmation dynamique. Chaque solution partielle évaluée lors de la recherche tabou est complétée grâce à la phase de retour arrière de la programmation dynamique. La valeur de la solution z donnée par (3.8) peut alors être obtenue comme suit.

$$cz = \sum_{j \in N''} c_j y_j + \max\left\{ v : \beta \geq \sum_{j \in N''} a_j y_j, (v, \beta) \in L \right\} \tag{3.9}$$

Cette utilisation de la programmation dynamique au sein de la recherche tabou justifie également le terme d'intensification globale. Un second niveau d'intensification étant ajouté dans l'algorithme tabou. L'algorithme 3.3 décrit le processus pour obtenir une solution complète sur l'ensemble N à partir d'une solution partielle sur N''.

Notons que d'un point de vue algorithmique la liste L est conservée triée lors de la phase de construction de manière à accélérer le processus de recherche de la solution x'. Le tri est réalisé au moment de la fusion des deux listes. L'opérateur « + » de la ligne 8 de l'algorithme 3.3 consiste à compléter la solution x'' par la solution x'.

Algorithme 3.3 Procédure de génération d'une solution sur N à partir de L et de x''

Entrées : le problème *SADM*; la liste L; la solution partielle x'' sur N''.
Sortie : la solution complète x sur N.

1 : Trouver (v^*, β^*, k^*) dans L tel que $v^* = \max\{v : (v, \beta, k) \in L | \beta \geq A'' x''\}$
2 : $x' = 0$;
3 : **tant que** $k^* > 0$ **faire**
4 : $x'_{k^*} = 1$;
5 : Trouver (v', γ, k') dans L tel que $v' = v^* - c_{k^*}$ et $\gamma = \beta - a_{k^*}$
6 : $(v^*, \beta^*, k^*) = (v', \gamma', k')$;
7 : **fin tant que**
8 : $x = x' + x''$;

Notons aussi que la programmation dynamique pourrait être remplacée par une autre méthode de résolution comme un algorithme de séparation et évaluations par exemple. L'avantage d'utiliser la programmation dynamique est que seul le second membre change dans tous les problèmes réduits à résoudre ici. Nous illustrons le déroulement de l'algorithme dans la section 3.3.

L'algorithme que nous avons mis en œuvre repose sur une liste tabou non stricte. Ce choix a été guidé par deux raisons majeures. La première est que la complexité spatiale et/ou temporelle associée à l'implémentation d'une liste stricte peut être important comme nous l'avons vu dans le chapitre 2, alors que la mise en place d'une liste non stricte est nettement moins coûteuse. De plus la complexité d'implémentation d'une liste stricte est également plus importante et finalement l'utilisation d'une liste non stricte a déjà permis l'obtention de bons résultats. Nous avons défini notre liste non stricte à l'aide d'un vecteur. Il suffit en effet de mémoriser le numéro de l'itération à partir de laquelle un mouvement est autorisé. Ce vecteur, noté *liste*, est de dimension *n''*. Le statut tabou d'un mouvement *i*, noté *mvt*(*i*), s'établit à une itération (*iter*) donnée comme suit

$$liste[i] > iter \Leftrightarrow mvt(i)\,\text{interdit} \quad \forall i \in N''.$$

La mise à jour du vecteur après un mouvement s'effectue alors de la manière suivante

$$liste[i] = iter + tenure\,.$$

La difficulté de l'utilisation de ce genre de liste est le réglage du paramètre *tenure* qui correspond au nombre d'itérations pendant lequel un mouvement est interdit. Plusieurs possibilités sont envisageables pour définir cette valeur à l'aide d'une fonction ou d'une constante par exemple. C'est généralement ce dernier cas qui est rencontré dans la littérature. Dans notre approche nous fixons la valeur de *tenure* de manière expérimentale (voir section 3.4). Nous définissons aussi un critère d'aspiration classique qui correspond au fait qu'une variable tabou peut être modifiée si cette modification permet d'améliorer la meilleure solution rencontrée depuis le début de la recherche.

Nous avons référencé dans le chapitre 2 quelques articles basés sur la recherche tabou dans lesquels la visite de solutions non réalisables est autorisée au cours du processus. Nous avons choisi dans notre cas de ne pas permettre la visite de solutions non réalisables. Ce choix s'explique par le principe de l'algorithme global lui-même. En effet lorsque nous effectuons un mouvement au sein de l'algorithme tabou, nous devons compléter la solution partielle

courante avec une solution partielle définie sur le sous-ensemble *N'*. La solution du problème associé à *N'* est fournie par la liste de programmation dynamique qui nous donne une solution optimale (et donc réalisable). Le fait d'accepter un mouvement non réalisable sur l'ensemble *N''* impliquerait donc d'avoir une solution nulle sur *N*', ce qui limiterait l'utilisation et la pertinence de la programmation dynamique au sein de notre processus d'intensification globale. Il serait toutefois bien entendu possible de définir plusieurs stratégies pour autoriser ce genre de mouvements.

Nous pourrions par exemple découper la recherche en différentes périodes. Pendant certaines d'entre elles la recherche tabou serait effectuée sur l'ensemble *N* des variables de décision et pourrait ainsi autoriser la visite de solutions non réalisables.

De manière à pouvoir mesurer l'impact du processus d'intensification globale sur les performances de la recherche tabou, nous avons comparé les résultats obtenus avec ceux d'un algorithme tabou simple appliqué à l'ensemble des variables *N* et basé sur des composants semblables à ceux utilisés dans le processus d'intensification globale.

L'algorithme tabou obtenu n'a pas été proposé dans le but d'améliorer les meilleurs résultats connus pour le problème *SADM* car il repose sur des concepts trop simples pour cela. La motivation de l'implémentation de cet algorithme est surtout liée à la comparaison des résultats avec le processus d'intensification globale. Le voisinage $V(x)$ d'une solution x est basé sur celui du processus d'intensification globale, et est défini par

$$V(x) = \left\{ y \in \{0;1\}^n \,\middle|\, Ay \le b, \sum_{j \in N} \left| x_j - y_j \right| = 1 \right\}.$$

La taille de ce voisinage est clairement égale à $|N|$. Notons que nous ne considérons ici que les solutions réalisables du problème. La fonction d'évaluation consiste à choisir à chaque itération le mouvement qui correspond à la meilleure amélioration lorsqu'il est possible d'améliorer la solution courante. Dans le cas contraire, c'est le mouvement qui dégrade le moins la valeur de l'objectif qui est choisi. Les autres composants de l'algorithme sont également basés sur ceux utilisés dans l'algorithme de recherche tabou du processus d'intensification globale.

3.2.2 Autres éléments de la recherche tabou

Plutôt que démarrer la recherche depuis une solution nulle, nous utilisons une heuristique basée sur la programmation linéaire (*HPL*) pour obtenir une solution réalisable

d'assez bonne qualité en peu de temps. Cette heuristique est présentée dans le chapitre 5 de ce mémoire. De manière à ce que les chapitres de ce mémoire soient le plus possible indépendants nous précisons ici rapidement les étapes de cette heuristique.

L'heuristique *HPL* consiste à résoudre dans un premier temps la relaxation en continu du problème et à en récupérer une solution optimale. Elle résout ensuite de manière optimale le problème réduit obtenu en fixant l'ensemble des variables entières de la solution précédente. Ce problème comporte un petit nombre de variables si le nombre de contraintes du problème est petit. L'application de cette heuristique permet d'appliquer l'algorithme de programmation dynamique à partir d'une solution de bonne qualité comme nous le montrons dans la section 3.4, et ainsi augmenter les performances en terme de réduction de l'algorithme.

Nous définissons les opérateurs d'intensification et de diversification sur l'ensemble N des variables de décision car ces opérateurs doivent être utilisés de manière globale pour prendre en compte l'ensemble des données du problème ainsi que l'ensemble de la mémoire associée à la recherche. L'intensification et la diversification sont deux éléments majeurs de la recherche tabou. Ils permettent, comme nous l'avons vu dans le chapitre 2, d'explorer une région de l'espace de recherche de manière plus approfondie si elle semble être une région prometteuse, ou au contraire de changer complètement de région à explorer si le processus à tendance à revenir trop souvent dans une partie de l'espace de recherche. Pour pouvoir appliquer des phases d'intensification dans des régions prometteuses, nous utilisons une population de solutions correspondant aux meilleurs optima locaux rencontrés au cours de la recherche. Ces solutions sont par définition de bonne qualité, et l'information qu'elles contiennent peut s'avérer utile pour guider la recherche vers une solution optimale du problème. Nous gardons *nb* solutions élites, avec *nb* paramètre du programme. La procédure d'intensification est lancée lorsque la meilleure solution rencontrée est améliorée, ou lorsque celle-ci n'a pas été améliorée pendant n/m itérations consécutives. Elle est appliquée à une des *nb* solutions. Une fois que la procédure d'intensification est terminée, la solution est supprimée de l'ensemble et elle sera remplacée par un nouvel optimum local rencontré au cours de la recherche. L'heuristique utilisée pour les phases d'intensification est la procédure *Complément* décrite dans l'algorithme 3.4. Cette heuristique repose sur un principe glouton. Nous retirons un objet de la solution courante et nous essayons de le remplacer par un ou plusieurs autres objets. Les objets à ajouter sont parcourus selon les rapports $c_j/\mu A^j$ pour chaque variable x_j, avec μ un multiplicateur de dimension m égal aux variables duales à

l'optimum de la relaxation en continu dans nos expérimentations. Une extension naturelle de cette heuristique consiste à mettre k variables (avec $k \leq |VF_1|$) à 0 dans la solution x simultanément au lieu d'une seule. Dans nos expérimentations nous avons utilisé la valeur k = 2. Notons également que cette heuristique peut être adaptée de manière à prendre en compte le statut tabou des mouvements.

Algorithme 3.4 Heuristique Complément

Entrées : le problème P; une solution réalisable x.
Sortie : une solution réalisable du problème z avec $cz \geq cx$.

1 : $z = x$;
2 : $VF_1 = \{j \in N : x_j = 1\}$; $VF_0 = \{j \in N : x_j = 0\}$;
3 : **pour** $j \in VF_1$ **faire**
4 : $y = x$;
5 : $y_j = 0$;
6 : **pour** $i \in VF_0$ **faire**
7 : $i^* = \arg\max\left\{\dfrac{c_j}{\mu A^i} \middle| i \in VF_0 \text{ et } Ay + A^i \leq b\right\}$;
8 : **si** i^* existe **alors** $y_{i^*} = 1$;
9 : **fin pour**
10 : **si** $cy > cz$ **alors** $z = y$;
11 : **fin pour**

La diversification est utilisée pour conduire la recherche dans des régions inexplorées ou peu explorées. Nous utilisons pour cela périodiquement l'information associée à la fréquence des variables modifiées au cours de la recherche. La fréquence fait partie de la mémoire à long terme du processus. Elle peut consister à conserver pour chaque variable le nombre total de modifications qu'elle a subi. Dans notre cas nous utilisons un tableau noté F dans lequel chaque valeur F_j correspond au nombre de fois que la variable x_j a été fixée à 1 depuis le début de la recherche. Cela nous permet de privilégier les variables les plus rarement fixées à '1' lorsque nous appliquons une phase de diversification. L'ordre dans lequel sont parcourues les variables est également important. Nous utilisons l'ordre défini par (3.10) qui dépend à la fois de la fréquence et de l'efficacité de la variable

$$\delta\left(\frac{c_j}{\mu A^j}\right) + (1-\delta)F_j \tag{3.10}$$

avec δ paramètre $\in [0,1]$, et A^j le vecteur des m poids associés à l'objet j.

Cette valeur nous permet d'avoir un équilibre entre la fréquence et l'efficacité de chaque variable. Ainsi lorsque δ tend vers 0 (resp. 1) nous favorisons la fréquence (resp.

l'efficacité). La valeur de δ est modifiée à chaque application de la diversification. Nous commençons par une valeur égale à 1 (car la fréquence n'est pas très significative au début de la recherche), et nous diminuons cette valeur de 0.1 jusqu'à arriver à 0. Nous repartons alors de la valeur 0.5. L'algorithme 3.5 de diversification génère une nouvelle solution obtenue à partir de la solution nulle en fixant à 1 les variables selon l'ordre des valeurs définies par (3.10). La solution obtenue doit cependant être réalisable.

Algorithme 3.5 Heuristique de diversification de la recherche

Entrées : le problème P; le vecteur des fréquences F; le paramètre pour le tri δ.
Sortie : une solution réalisable x de P.

1 : $x = 0$; $J = N$;
2 : **pour** tout j allant de 1 à $|J|$ **faire**
3 : $j^* = \arg\max\left\{\delta\left(\frac{c_j}{\mu A^j}\right) + (1-\delta)F_j \,|\, j \in J\right\}$;
4 : **si** $Ax + A^{j^*} \leq b$ **alors** $x_{j^*} = 1$;
5 : $J = J - \{j^*\}$;
6 : **fin pour**

Ce genre d'approche consiste à pénaliser certains mouvements en fonction de la mémoire de la recherche et a été évoqué par Glover et Laguna [GL97]. Ce type de diversification peut produire plusieurs fois la même solution si la fréquence n'a pas suffisamment évolué entre deux applications consécutives. Il est cependant facile d'éviter une telle situation en modifiant par exemple un nombre aléatoire de variables dans la solution obtenue, ou en changeant la valeur du paramètre δ. Une phase de diversification est activée dès que K phases d'intensification consécutives n'améliorent pas la solution en entrée.

3.3 Recherche tabou avec programmation dynamique

Nous présentons dans cette section le fonctionnement du processus d'intensification globale (*IG*) obtenu en créant la coopération entre les algorithmes de programmation dynamique et de recherche tabou. Nous illustrons aussi le déroulement de notre méthode sur une instance de petite taille de problème de *SADM*. Nous pouvons résumer notre méthode hybride par les quatre phases suivantes.

- Phase 1 : Initialisation. Une solution réalisable du problème est générée à l'aide d'une heuristique adaptée au problème. L'ensemble des variables N du problème est partitionné en deux sous-ensembles notés N' et N''.

- Phase 2 : Programmation dynamique. L'ensemble des problèmes réduits associé au sous-ensemble *N'* est résolu de manière optimale. Nous appliquons pour cela un algorithme de programmation dynamique avec une liste *L* pour récupérer les solutions optimales de ces sous-problèmes pour lesquels seul le membre droit change. Durant ce processus de programmation dynamique des règles de réduction sont appliquées pour fixer des variables à leur valeur optimale. Le processus est répété dans le but de fixer le plus de variables possibles.
- Phase 3 : Recherche tabou. La recherche tabou est appliquée sur l'espace de recherche défini par le sous-ensemble *N''*. L'algorithme utilise la liste *L* générée précédemment pour explorer l'espace de recherche et améliorer la solution générée durant la phase 2.
- Phase 4 : Récurrence. Les phases 2 et 3 sont relancées tant que la solution obtenue est améliorée et/ou qu'au moins une variable peut être fixée.

Le schéma de cette méthode peut être appliqué à n'importe quel problème d'optimisation combinatoire pour lequel il existe une approche efficace basée sur la programmation dynamique. Nous illustrons le déroulement du processus d'intensification globale sur l'instance suivante de problème de sac-à-dos multidimensionnel avec $n = 10$ et $m = 3$.

$$(GK1)\quad \begin{bmatrix} \max & 20x_1 + & 18x_2 + & 15x_3 + & 14x_4 + & 12x_5 + & 9x_6 + & 7x_7 + & 5x_8 + & 3x_9 + & 2x_{10} & \\ s.c. & 15x_1 + & 16x_2 + & 12x_3 + & 12x_4 + & 10x_5 + & 10x_6 + & 8x_7 + & 5x_8 + & 4x_9 + & 3x_{10} & \le 45 \\ & 22x_1 + & 21x_2 + & 16x_3 + & 14x_4 + & 15x_5 + & 7x_6 + & 5x_7 + & 2x_8 + & 4x_9 + & 4x_{10} & \le 50 \\ & 18x_1 + & 20x_2 + & 15x_3 + & 10x_4 + & 9x_5 + & 8x_6 + & 2x_7 + & 6x_8 + & 2x_9 + & 5x_{10} & \le 40 \\ & x_j \in \{0,1\} & j = 1,\ldots,10 & & & & & & & & & \end{bmatrix}$$

La valeur optimale de cette instance est $v(GK1) = 50$.

Phase 1 – Initialisation : La valeur de la relaxation en continu est égale à 51.60, et une solution optimale est $\bar{x}$ = (0.90 0 0 1 0.58 0.07 1 1 0 0).

A partir de la solution $\bar{x}$, l'heuristique *HPL* résout le problème réduit (*P*1) suivant.

$$(P1)\quad \begin{bmatrix} \max & 20x_1 + & 12x_5 + & 9x_6 & \\ s.c. & 15x_1 + & 10x_5 + & 10x_6 & \le 20 \\ & 22x_1 + & 15x_5 + & 7x_6 & \le 29 \\ & 18x_1 + & 9x_5 + & 8x_6 & \le 22 \\ & x_1, x_5, x_6 & \in \{0,1\} & & \end{bmatrix}$$

La valeur optimale du problème (*P*1) vaut 21 et correspond à la solution (0 1 1). La solution initiale obtenue pour le problème *GK*1 est donc

$$x^0 = (0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 0\ 0) \text{ avec } cx^0 = 47.$$

L'heuristique *Complément* (algorithme 3.4) est utilisée pour améliorer la solution x^0. Elle produit une nouvelle solution réalisable

$$x^0 = (0\ 0\ 1\ 0\ 1\ 1\ 1\ 1\ 0\ 0) \text{ avec } cx^0 = 48.$$

Les variables x_j sont triées dans l'ordre décroissant des bornes supérieures u_j, avec u_j la valeur de la relaxation en continu du problème $SADM(\text{e-}x^0, \{j\})$.

u_3	u_4	u_1	u_6	u_5	u_8	u_9	u_7	u_{10}	u_2
51.60	51.60	51.60	51.50	51.36	51.11	50.82	49.13	49.06	48.65

Phase 2 – Programmation dynamique : La valeur de *n'* est fixée à 4 dans nos expérimentations, avec $N' = \{3, 4, 1, 6\}$. La table suivante donne la liste *L* générée durant les quatre premières itérations de la procédure de programmation dynamique (rappelons que *v* dénote la valeur de la solution, et b_i-$a_i x$ la capacité résiduelle associée à la $i^{\text{ième}}$ contrainte). Pour simplifier la présentation nous ne précisons par l'élément « *k* » dans les éléments de la liste car il n'apporte pas d'information importante pour la compréhension.

k	$L = \{(v;\ b_1\text{-}a_1x, b_2\text{-}a_2x, b_3\text{-}a_3x)\}$
1	(0; 45, 50, 40), (15; 33, 34, 25)
2	(0; 45, 50, 40), (14; 33, 36, 30), (15; 33, 34, 25), (29; 21, 20, 15)
3	(0; 45, 50, 40), (14; 33, 36, 30), (15; 33, 34, 25), (20; 30, 28, 22) (29; 21, 20, 15), (34; 18, 14, 12), (35; 18, 12, 7)
4	(0; 45, 50, 40), (9; 35, 43, 32), (14; 33, 36, 30), (15; 33, 34, 25) (20; 30, 28, 22),(23; 23, 29, 22), (24; 23, 27, 17),(29; 20, 21, 14), (29; 21, 20, 15), (34; 18, 14, 12), (35; 18, 12, 7), (38; 11, 13, 7), (43; 8, 7, 4)

La borne inférieure l_4 sur la valeur optimale de (*GK*1) est calculée selon (3.4)

$$l_4 = 12x_5^0 + 5x_8^0 + 3x_9^0 + 7x_7^0 + 2x_{10}^0 + 18x_2^0 + \max\left\{v \,\middle|\, \beta \geq \sum_{j \in N'} a_j x_j^0, (v, \beta) \in L\right\}$$

$$l_4 = 24 + \max\{v \mid \beta \geq (23, 22, 17), (v, \beta) \in L\} = 24 + 24 = 48.$$

En effet, $\max\{v \mid \beta \geq (23, 22, 17), (v, \beta) \in L\}$ correspond à l'élément (24; 23, 27, 17).

De plus, comme $l_4 = 48 \geq \lfloor u_{10} \rfloor = 48$, la variable x_2 est fixée à 0 ($x_2^0 = 0$). Lorsque nous relançons l'algorithme de programmation dynamique sur le problème réduit, aucun changement ne se produit.

Phase 3 – Recherche tabou : L'algorithme de recherche tabou est appliqué sur le sous-ensemble $N'' = N - N' = \{5, 7, 8, 9, 10\}$ et fournit la solution réalisable suivante dans $\{0,1\}^n$, obtenue en quatre mouvements

$$x^* = (1\ 0\ 0\ 1\ 0\ 1\ 1\ 0\ 0\ 0) \text{ avec } cx^* = 50.$$

Pour illustrer la procédure de génération de la solution partielle sur N' complémentaire d'une solution courante sur N'' décrite par l'algorithme 3.3, soit $x'' = (x_5, x_7, x_8, x_9, x_{10}) = (1\ 1\ 1\ 0\ 0)$ la solution courante sur N''. Après le mouvement $x''_5 = 0$, la solution x'' devient (0 1 1 0 0). Ce mouvement libère les capacités (10, 15, 9). L'étape 1 de la procédure conduit à l'élément (29; 21, 20, 15) dans la liste L, et l'étape 2 génère la solution optimale sur N' suivante $x' = (x_3, x_4, x_1, x_6) = (1\ 1\ 0\ 0)$. La solution complète visitée est donc égale à $x = (x_1, x_3, \ldots, x_{10}) = (0\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 0)$, avec $cx = 41$ $(= 29 + c''x'')$.

Comme la solution finale de la recherche tabou est meilleure que la solution initiale, le processus est relancé avec cette solution comme nouvelle solution initiale.

Phase 2 – Programmation dynamique : Le processus est relancé avec x^* comme solution initiale et $N := N - \{2\}$ puisque la variable x_2 est fixée à 0. Les bornes supérieures u_j sont désormais comme suit :

u_8	u_6	u_5	u_9	u_3	u_1	u_4	u_7	u_{10}
51.60	51.50	50.96	50.82	50.82	50.80	50.31	49.13	49.06

L'algorithme de programmation dynamique est appliqué sur le sous-ensemble $N' = \{8, 6, 5, 9\}$, et la borne inférieure l_4 (correspondant à x_9) est égale à 50. Comme $l_4 \geq u_5$ (où u_5 est associée à la variable x_3 du problème original), l'algorithme s'arrête comme précisé dans la proposition 3.1 et x^* est une solution optimale de ($GK1$).

3.4 Résultats numériques

Tous les algorithmes présentés dans ce chapitre ont été codés en langage 'C' et compilés avec « gcc » et l'option –O2. La librairie CPLEX est utilisée pour les calculs des bornes supérieures et pour la résolution du problème réduit lors de la phase initiale. L'ensemble des tests a été réalisé sur une station de travail Ultra Sparc Solaris 2.6 300MHz avec 128Mo de RAM.

L'algorithme que nous proposons a été testé sur un ensemble conséquent d'instances de problèmes de *SADM*. Nous avons utilisé un premier ensemble de 270 instances corrélées et difficiles de la OR-Librairie disponible sur Internet [Bea90]. Ces instances ont été générées

selon la procédure proposée par Fréville et Plateau [FP94]. Pour toutes ces instances les coefficients a_{ij} sont des nombres entiers uniformément générés dans $U(0,1000)$. Les membres droits des contraintes (b_i) sont calculés selon la formule $b_i = \alpha \sum_{j \in N} a_{ij}$ avec $\alpha = 0.25, 0.5, 0.75$. Les coefficients associés à l'objectif du problème (c_j) sont corrélés aux coefficients a_{ij} comme suit : $c_j = \sum_{i \in M} a_i / m + 500\delta_j, j \in N$, avec δ_j un réel uniformément généré dans $U(0,1)$. Les 270 instances ont été générées en changeant le nombre de contraintes (m = 5, 10, 30), de variables (n = 100, 250, 500), et la valeur de α (α = 0.25, 0.5, 0.75). Finalement 10 instances distinctes ont été produites pour chaque combinaison de n-m-α.

Nous avons également testé notre approche sur un ensemble de 360 instances générées aléatoirement. Ces instances ont un nombre de variables plus important (n = 500, 1000, 2000 et 5000), et ont toutes 5 contraintes. Nous utilisons comme pour les instances précédentes un paramètre α (α = 0.25, 0.5, 0.75). Les coefficients a_{ij} sont des nombres entiers générés uniformément dans $U(0,1000)$ les coefficients c_j sont corrélés aux coefficients a_{ij} de la manière suivante :

- non corrélés : $c_j \in U(0,1000)$.
- moyennement corrélés : $c_j = \frac{1}{m} \sum_{i=1}^{m} a_{ij} + \xi$ avec $\xi \in U(-100,100)$.
- fortement corrélés : $c_j = \frac{1}{m} \sum_{i=1}^{m} a_{ij} + 100$.

Finalement nous générons 30 instances distinctes pour chaque combinaison n – degré de corrélation – α.

Nous avons utilisé un dernier ensemble de 18 instances dues à Glover et Kochenberger [GK96]. Le nombre de variables varie entre 100 et 2500, et le nombre de contraintes entre 15 et 100. Ces instances sont connues comme étant difficiles à résoudre par des méthodes de séparation et évaluations.

Comme nous l'avons dit dans la section 3.1, le calcul de n' est effectué de manière expérimentale. Il est cependant possible d'estimer la valeur de ce paramètre. En effet supposons que l'algorithme reste rapide tant que la taille totale de la liste L n'excède pas *Mem* unités. Durant la phase de construction de la liste de programmation dynamique, le nombre d'éléments dans la liste L double à chaque itération dans le pire des cas. Ainsi après avoir fait n' itérations, la liste L est formée d'au plus $2^{n'}$ éléments, avec chaque élément de L qui correspond à une solution réalisable du problème. Comme chaque élément est composé de m

+ 2 valeurs, la taille de L est au plus de $2^{n'} \times (m+2)$ après les n' itérations. Une valeur de n' égale à $log_2 Mem - log_2 (m+2)$ semble donc être une valeur permettant à l'algorithme de rester rapide. La valeur *Mem* reste bien entendu à déterminer de manière expérimentale en fonction de la machine utilisée. Dans nos expérimentations la valeur de n' est petite par rapport à n (i.e. entre 0.3% et 15%).

Avant de présenter les résultats obtenus nous précisons les choix des différents paramètres des algorithmes ainsi que les critères de comparaison choisis. Le nombre maximal d'itérations effectuées par l'algorithme de recherche tabou au sein du processus d'intensification globale est calculé en fonction de la taille de N'' avec une borne inférieure et une borne supérieure (entre 1000 et 5000 itérations pour être plus précis). Les valeurs des paramètres *nb* et K sont respectivement comprises entre 5 et 20, et 5 et m. Finalement la valeur du paramètre *tenure* a été fixée à m. Nous avons testé différentes valeurs pour ce paramètre mais aucune version ne domine toutes les autres. Une des méthodes envisagée consiste à initialiser la valeur de la tenure à m, puis à augmenter la valeur associée à une variable à chaque fois que celle-ci est modifiée au cours de la recherche. L'inconvénient de cette approche est qu'elle introduit un second paramètre pour limiter l'augmentation de la longueur de la liste tabou. Elle permet cependant de définir une méthode *pseudo dynamique* de sa mise à jour. Les résultats préliminaires obtenus sont cependant dans l'ensemble moins bons qu'en utilisant la valeur statique m.

Nous comparons la qualité de nos solutions finales avec celles obtenues par CPLEX et par l'algorithme de recherche tabou seul (décrit dans la section 3.2). Le logiciel CPLEX et cet algorithme tabou ont été lancés avec un temps d'exécution identique à celui mis par le processus d'intensification globale. Ainsi par exemple pour les instances de taille $n = 10$ et $m = 5$ dans le tableau 3.1 le temps alloué aux deux algorithmes est de 25 secondes. La qualité d'une solution obtenue par le processus d'intensification globale (*IG*) est déterminée par la formule : ((valeur_CPLEX – valeur_IG) / valeur_CPLEX)) * 100. Les valeurs présentes dans les colonnes *%cx* sont obtenues par un calcul similaire. Notons aussi que tous les temps d'exécution mentionnés dans les tableaux (colonnes *CPU*) sont en secondes. Nous avons choisi de faire une comparaison avec les solutions obtenues par le logiciel CPLEX dans le but de montrer que le processus d'intensification globale que nous proposons permet de rivaliser avec un logiciel d'optimisation performant lorsque le temps d'exécution des deux approches est le même. Nous aurions également pu nous comparer aux résultats obtenus par Vasquez et Hao [VH01b] ou Vasquez et Vimont [VV05], mais les approches décrites dans ces deux

articles ont des temps d'exécution nettement supérieurs à ceux de notre algorithme et ont été proposés spécifiquement pour résoudre le problème *SADM* (voir chapitre 2 section 2.3.2).

Nous présentons dans le tableau 3.1 les résultats obtenus sur les instances de la OR-Librairie. Pour chaque classe d'instances ($n - m$), nous donnons la taille du sous-ensemble N' associé à la programmation dynamique, la qualité moyenne de la solution et le temps d'exécution de la phase initiale (*SI*), de la programmation dynamique (*PD*) et du processus d'intensification globale (*IG*). Nous donnons aussi la qualité de la solution obtenue par la recherche tabou seule (*RT*) et le pourcentage de variables fixées après le premier (resp. dernier) passage dans l'algorithme de programmation dynamique, %*F*1 (resp. *%F*2). Chaque ligne est une moyenne sur 30 instances. Pour la colonne *IG* la valeur *%cx* correspond à la qualité de la solution finale de l'approche, alors que le temps d'exécution mentionné est le temps total des applications de la recherche tabou.

Problème			*SI*		*PD*		*Reduction*		*IG*		*RT*
n	*m*	|*N*'|	*%cx*	*CPU*	*%cx*	*CPU*	*%F1*	*%F2*	*%cx*	*CPU*	*%cx*
100			0.34	0	0.08	0.1	25.5	32.3	0.01	22.5	0.01
250	5	15	0.15	0	0.10	0.1	30.2	49.9	0.01	65.2	0.03
500			0.06	0	0.04	0.1	39.2	57.8	0.01	138.4	0.05
100			0.55	0	0.27	0.1	5.1	8.6	0.06	38.9	0.02
250	10	15	0.24	0	0.17	0.1	4.7	11.9	0.06	113.7	0.09
500			0.11	0.1	0.08	0.1	5.9	13.5	0.03	414.8	0.12
100			0.27	0.9	0.26	0.9	0.2	0.3	0.13	31.1	0.02
250	30	13	0.10	2.8	0.10	2.8	0	0	0.07	109.7	0.09
500			0.04	6.0	0.04	6.4	0	0	0.03	339.4	0.04
Pire des cas			0.55	6.0	0.27	6.4	0	0	0.13	414.8	0.12
Moyenne			0.21	1.1	0.13	1.2	12.3	19.4	0.046	141.5	0.053

Tableau 3.1 – Résultats obtenus sur les instances de la OR-Librairie.

Nous pouvons voir dans le tableau 3.1 que la phase initiale permet l'obtention d'une solution d'assez bonne qualité en un temps raisonnable (une seconde en moyenne et six secondes dans le pire des cas). La qualité de cette solution s'améliore avec n et se dégrade lors du passage de $m = 5$ à 10 avant de s'améliorer à nouveau entre $m = 10$ et $m = 30$.

La programmation dynamique permet d'améliorer nettement la solution initiale pour les instances comportant 5 ou 10 contraintes. Pour les plus grandes instances elle parvient à améliorer la solution initiale rarement. Cela s'explique par le pourcentage de variables fixées à leur valeur optimale qui est intéressant pour les instances comportant peu de contraintes, mais très faible dans le cas $m = 30$. Notons l'impact positif de la récursivité du processus sur la fixation en comparant les valeurs des colonnes *%F*2 et *%F*1 pour les autres instances.

La recherche tabou permet d'obtenir des solutions finales très proches des solutions obtenues par CPLEX. Pour être plus précis nous obtenons 97 fois la même solution que CPLEX et 26 fois une meilleure solution que CPLEX sur les 270 instances de l'ensemble (soit 46% des instances pour lesquelles notre solution est au moins aussi bonne). Le processus d'intensification globale est finalement le moins performant pour les instances comportant 100 variables et 10 ou 30 contraintes. L'algorithme tabou seul obtient 21 meilleures solutions que le processus d'intensification globale (7.8%), en particulier sur ces instances. La performance de cet algorithme peut s'expliquer par le grand nombre de mouvements effectués sur ces instances par rapport au nombre de mouvements de la recherche tabou au sein de l'intensification globale. Il y a en effet une nette différence du temps d'exécution associé à l'évaluation du voisinage entre les instances comportant 100 variables et les autres.

Notons pour terminer que le temps total de l'algorithme d'intensification globale n'est pas excessif sur ces instances. Nous sommes en effet dans le pire des cas de l'ordre de 7 minutes, ce qui nous permet d'obtenir un processus rapide pour générer des solutions de bonne qualité. Le temps total de l'approche dépend en partie du nombre d'exécutions de l'algorithme tabou. Il est en moyenne de 1.7 fois sur ces instances. Le processus d'intensification globale semble donc plus efficace lorsque le nombre de contraintes du problème est petit. C'est pourquoi les instances générées aléatoirement comportent toutes cinq contraintes. Cela nous permet de vérifier l'efficacité de l'approche lorsque le nombre de variables augmente.

Nous présentons dans le tableau 3.2 les résultats obtenus sur l'ensemble de 360 instances générées aléatoirement.

Problème		*SI*		*PD*		*Reduction*		*IG*		*RT*
n	*type* c_j	*%cx*	*CPU*	*%cx*	*CPU*	%F1	%F2	*%cx*	*CPU*	*%cx*
500	non	0.050	0.0	0.028	0.6	74.7	81.3	0.002	39.3	0.021
	faible	0.057	0.0	0.057	0.9	17.3	37.3	0.011	244.3	0.060
	fort	0.063	0.0	0.023	1.1	1.0	5.3	-0.002	307.7	0.020
1000	non	0.023	0.1	0.007	1.7	79.3	84.0	0.001	79.3	0.017
	faible	0.020	0.1	0.013	2.6	22.0	46.0	-0.001	829.0	0.037
	fort	0.017	0.1	0.001	3.8	1.0	4.7	-0.010	1463.0	-0.005
2000	non	0.008	0.2	0.007	6.0	81.0	87.3	0.000	211.7	0.011
	Faible	0.005	0.2	0.004	9.6	33.0	51.7	-0.002	1813.3	0.017
	Fort	0.005	0.3	-0.001	14.5	2.3	7.3	-0.006	1605.0	-0.006
5000	Non	0.002	1.0	0.002	35.4	83.3	89.0	-0.017	803.7	0.005
	Faible	-0.002	1.5	-0.002	47.3	47.0	55.3	-0.003	1612.0	0.002
	Fort	-0.002	1.8	-0.003	86.9	2.7	4.0	-0.005	1428.7	-0.005
Pire des cas		0.063	1.8	0.057	86.9	1.0	4.0	0.011	1813.3	0.060
Moyenne		0.014	0.7	0.009	31.0	31.0	39.5	-0.004	1244.9	0.013

Tableau 3.2 – Résultats obtenus sur les instances générées aléatoirement.

Nous précisons dans le tableau 3.2 les résultats pour chaque classe d'instances (*n* – degré de corrélation). La valeur de |*N'*| est ici fixée à 15 pour toutes les instances puisque le nombre de contraintes est constant. La présentation des résultats est identique au tableau précédent. Une différence notable dans ce tableau est l'apparition de chiffres négatifs dans les colonnes *%cx*. Nous parvenons en effet sur ces instances à obtenir de meilleures solutions en moyenne que celles fournies par CPLEX.

La phase initiale permet d'obtenir de meilleures solutions que celles de CPLEX pour les instances les plus difficiles (*n* = 5000 avec faible ou fort degré de corrélation). Le temps d'exécution de la phase initiale reste très faible ici. La programmation dynamique améliore la qualité de la solution précédente dans la majorité des cas. Notons que le pourcentage de réduction diminue logiquement avec le degré de corrélation des instances. Il augmente par contre la plupart du temps avec *n*. Le temps total passé dans l'algorithme de programmation dynamique reste peu élevé puisqu'il s'élève au plus à environ 90 secondes pour les instances les plus difficiles. Ici encore le temps d'exécution augmente logiquement avec le degré de corrélation et la taille des instances. La solution finale obtenue par le processus d'intensification globale est cette fois nettement meilleure que celle de l'algorithme tabou seul. Ces résultats montrent l'impact positif du processus d'intensification globale. Nous obtenons au total 268 solutions au moins aussi bonnes que celles de CPLEX sur cet ensemble d'instances (environ 74%). L'algorithme tabou seul domine le processus d'intensification globale sur 38 instances (environ 11%). Les résultats globaux montrent cependant qu'il est nettement moins robuste que le processus d'intensification globale sur les grandes instances. Le temps passé dans l'algorithme tabou est raisonnable vu la taille des instances. Notons que nous exécutons en moyenne 2.1 fois cet algorithme ici. Nous parvenons donc à améliorer plus facilement notre solution que précédemment.

Ces résultats montrent clairement l'efficacité et la robustesse du processus d'intensification globale pour la résolution de problèmes comportant un grand nombre de variables et peu de contraintes.

Le tableau 3.3 suivant permet d'affiner la comparaison entre l'algorithme de recherche tabou seul et le processus d'intensification globale sur les instances générées aléatoirement. Nous donnons les résultats obtenus par degré de corrélation des instances et ensuite par nombre de variables. Pour chaque catégorie nous précisons le nombre total d'instances, puis nous donnons la qualité moyenne de la solution finale par rapport à CPLEX (*%cx*), le nombre

d'égalité entre les solutions (*#égalité*) et le nombre de fois où l'algorithme obtient une meilleure solution que CPLEX (*#meilleur*).

Type	Nombre d'instances	Résultats	*IG*	*RT*
Non corrélés	120	*%cx*	-0.004	0.013
		#égalité	34	8
		#meilleur	45	4
Faiblement corrélés		*%cx*	0.001	0.029
		#égalité	7	1
		#meilleur	71	7
Fortement corrélés		*%cx*	-0.005	0.002
		#égalité	2	1
		#meilleur	109	100
n = 500	90	*%cx*	0.003	0.035
		#égalité	29	9
		#meilleur	29	16
n = 1000		*%cx*	-0.003	0.017
		#égalité	9	0
		#meilleur	57	28
n = 2000		*%cx*	-0.002	0.006
		#égalité	3	0
		#meilleur	60	31
n = 5000		*%cx*	-0.009	0.001
		#égalité	2	1
		#meilleur	79	36

Tableau 3.3 – Comparaison entre *IG* et *RT* pour les instances générées aléatoirement.

Comme nous l'avons dit précédemment, le processus d'intensification globale domine clairement l'algorithme tabou seul. Il est intéressant de noter dans le tableau 3.3 la différence des performances entre les deux algorithmes lorsque n augmente. La qualité moyenne de la solution finale sur les instances fortement corrélées est également assez différente. Pour les instances moins corrélées c'est surtout le nombre de solutions identiques ou meilleures que celles de CPLEX qui montre l'impact positif de l'intensification globale.

Nous présentons dans le tableau 3.4 les résultats obtenus sur les 18 dernières instances proposées par Glover et Kochenberger [GK96]. Cet ensemble peut lui-même être divisé en deux sous-ensembles. Le premier comporte 7 instances (*GK*18 à *GK*24) et le second 11 (*MK_GK*01 à *MK_GK*11). Nous précisons pour chaque instance le nombre de variables et de contraintes. Nous donnons ensuite la taille du sous-ensemble *N'*, puis la valeur de la solution obtenue par CPLEX, et pour chaque phase de l'algorithme la qualité de la solution et le temps d'exécution (colonnes *%cx* et *%CPU*). La dernière colonne donne la qualité de la solution obtenue par l'algorithme tabou seul.

Problème				*CPLEX*	*SI*		*DP*		*IG*		*RT*
Nom	*n*	*m*	*|N'|*		*%cx*	*CPU*	*%cx*	*CPU*	*%cx*	*CPU*	*%cx*
MK_GK01	100	15	14	3766	0.35	0.1	0.19	0.2	0.13	11.9	0.19
GK18	100	25	14	4522	0.04	0.9	0.04	1.5	0.02	16.2	-0.04
GK19	100	25	14	3867	0.18	0.5	0.18	0.7	0.10	15.1	0.18
GK20	100	25	14	5177	0.04	0.8	0.04	0.9	0.02	13.2	0.02
GK21	100	25	14	3199	0.16	0.6	0.16	0.8	0.13	14.0	0.16
GK22	100	25	14	2521	0.16	0.3	0.16	0.4	0.12	13.4	0.16
MK_GK02	100	25	14	3957	0.20	0.5	0.05	0.4	-0.03	11.2	0.20
MK_GK03	150	25	14	5649	0.16	1.4	0.14	0.7	0.00	23.3	0.16
MK_GK04	150	50	13	5763	-0.02	59.3	-0.02	1.9	-0.02	14.3	-0.02
GK23	200	15	14	9233	0.08	0.2	0.08	1.1	0.03	24.2	0.08
MK_GK05	200	25	14	7558	0.07	1.1	0.07	1.0	0.04	29.3	0.07
MK_GK06	200	50	13	7666	-0.07	177.4	-0.07	3.2	-0.07	21.1	-0.07
GK24	500	25	14	9064	0.08	0.8	0.08	2.5	0.07	139.7	0.11
MK_GK07	500	25	14	19209	0.02	0.6	0.02	5.3	-0.02	137.9	0.02
MK_GK08	500	50	13	18789	0.09	200	0.07	15.8	-0.05	111.9	-0.04
MK_GK09	1500	25	14	58080	0.01	1.5	0.00	36.1	-0.01	900.8	0.01
MK_GK10	1500	50	13	57277	0.24	200	0.24	126.4	-0.02	908.5	-0.01
MK_GK11	2500	100	12	95209	0.05	200	0.05	1943.3	-0.01	929.2	-0.01

Tableau 3.4 – Résultats sur les instances de Glover et Kochenberger.

Le processus de réduction ne permet pas de fixer de variables sur cet ensemble d'instances. Cela confirme les résultats précédents puisque ces instances sont difficiles et comportent un nombre de contraintes plus important. La qualité de la solution initiale varie fortement sur ces 18 instances. Elle peut être très éloignée de la solution finale de CPLEX comme pour *MK_GK*01 par exemple, ou meilleure que la solution finale de CPLEX pour les problèmes *MK_GK*04 et *MK_GK*06. Le temps total associé à cette phase initiale augmente logiquement avec la taille des instances et surtout avec *m*. Nous stoppons l'heuristique *HPL* après 200 secondes au maximum. Nous considérons en effet qu'il ne s'agit pas de la phase la plus importante au sein de l'intensification globale. La phase de programmation dynamique parvient dans la majorité des cas à améliorer la qualité de la solution initiale, et le temps total d'exécution de cet algorithme reste dans des proportions raisonnables. Seule la dernière instance qui est vraiment de grande taille pose problème à la programmation dynamique. La qualité de la solution finale est bonne : 8 fois supérieure à CPLEX, identique pour une instance et moins bonne pour les 9 dernières instances. Les meilleurs résultats par rapport à CPLEX sont une nouvelle fois obtenus lorsque la taille des instances augmente. L'algorithme tabou seul est moins efficace, excepté pour l'instance *GK*18 pour laquelle il fait mieux que CPLEX. Cet algorithme obtient les mêmes solutions que l'algorithme d'intensification globale pour les instances *MK_GK*04 et *MK_GK*06. Les solutions obtenues pour les plus grandes instances sont également très proches des solutions du processus d'intensification globale.

Nous terminons cette section par quelques observations supplémentaires liées à quelques tests complémentaires. De la même manière que nous avons montré l'impact du processus d'intensification globale par rapport à l'algorithme de recherche tabou seul, nous pouvons nous demander quel est l'impact de cette approche par rapport à la programmation dynamique utilisée seule comme une heuristique. Nous avons testé l'algorithme de programmation dynamique (avec récurrence) seul sur un sous-ensemble *N'* de plus grande taille pour les instances de la OR-Librairie. Sur la machine que nous avons utilisée nous pouvons par exemple fixer la valeur de $|N'|$ à 20 pour $m = 5$ et 10, et à 18 pour $m = 30$. Les résultats obtenus sont résumés dans le tableau 3.5. Pour chaque classe d'instances nous donnons dans la colonne *%F_PD* (resp. *%F_IG*) le pourcentage total de variables fixées à la fin de l'algorithme de programmation dynamique exécuté seul (resp. à la fin du processus d'intensification globale). Nous donnons de même la qualité de la solution finale obtenue par la programmation dynamique dans la colonne *%cx_PD* et nous rappelons celle de l'intensification globale dans *%cx_IG*. Finalement, la colonne *CPU_PD* précise le temps total de l'algorithme de programmation dynamique exécuté seul.

Problème			Fixation		Solution		*CPU_PD*
n	*m*	*n'*	*%F_PD*	*%F_IG*	*%cx_PD*	*%cx_IG*	
100			29.7	32.3	0.03	0.005	3
250	5	20	39.5	49.9	0.05	0.01	3
500			48	57.8	0.02	0.01	3.5
100			5.4	8.6	0.18	0.06	3.2
250	10	20	8.9	11.9	0.12	0.06	3.5
500			8	13.5	0.07	0.03	4.6
100			0.2	0.3	0.24	0.13	2
250	30	18	0	0	0.10	0.07	4.4
500			0	0	0.04	0.03	9.9

Tableau 3.5 – Comparaisons entre *IG* et *PD* seule pour la OR-Librairie**.**

Le temps d'exécution total de l'algorithme de programmation dynamique reste faible puisqu'il est de l'ordre de 10 secondes sur les plus grandes instances. Les ressources mémoires nécessaires à cet algorithme nous empêchent cependant d'augmenter la valeur de *n'*. La qualité finale des solutions est néanmoins assez nettement inférieure à celle obtenue avec le processus d'intensification globale. Notons aussi le gain intéressant en terme de fixation de variables avec le processus d'intensification globale.

Le fait d'augmenter la valeur de *n'* rend la liste de programmation dynamique difficilement utilisable par l'algorithme tabou. La recherche de la solution partielle devient en effet beaucoup plus longue. Nous pouvons cependant envisager de modifier la valeur de *n*' d'une unité par exemple. Nous parvenons alors à une très légère amélioration de la qualité

moyenne de nos solutions. Le pourcentage moyen de variables fixées augmente également un peu (de l'ordre de 1.8%). L'augmentation importante du temps d'exécution total moyen de l'algorithme ne semble cependant pas justifier ce choix puisque nous passons d'un temps total moyen de 140 secondes à 400 secondes.

Nous illustrons pour terminer avec la figure 3.1 l'influence de la valeur de *n'* sur la qualité de la solution finale et le temps d'exécution du processus d'intensification globale sur un exemple. L'instance appartient à la classe $n = 250$, $m = 5$. Nous avons exécuté notre algorithme avec les valeurs de *n'* égales à 10, 12, 14, 16 et 18. Nous donnons dans la figure la valeur de la solution obtenue (courbe avec des carrés) ainsi que le temps total de l'algorithme en secondes (courbe avec des triangles).

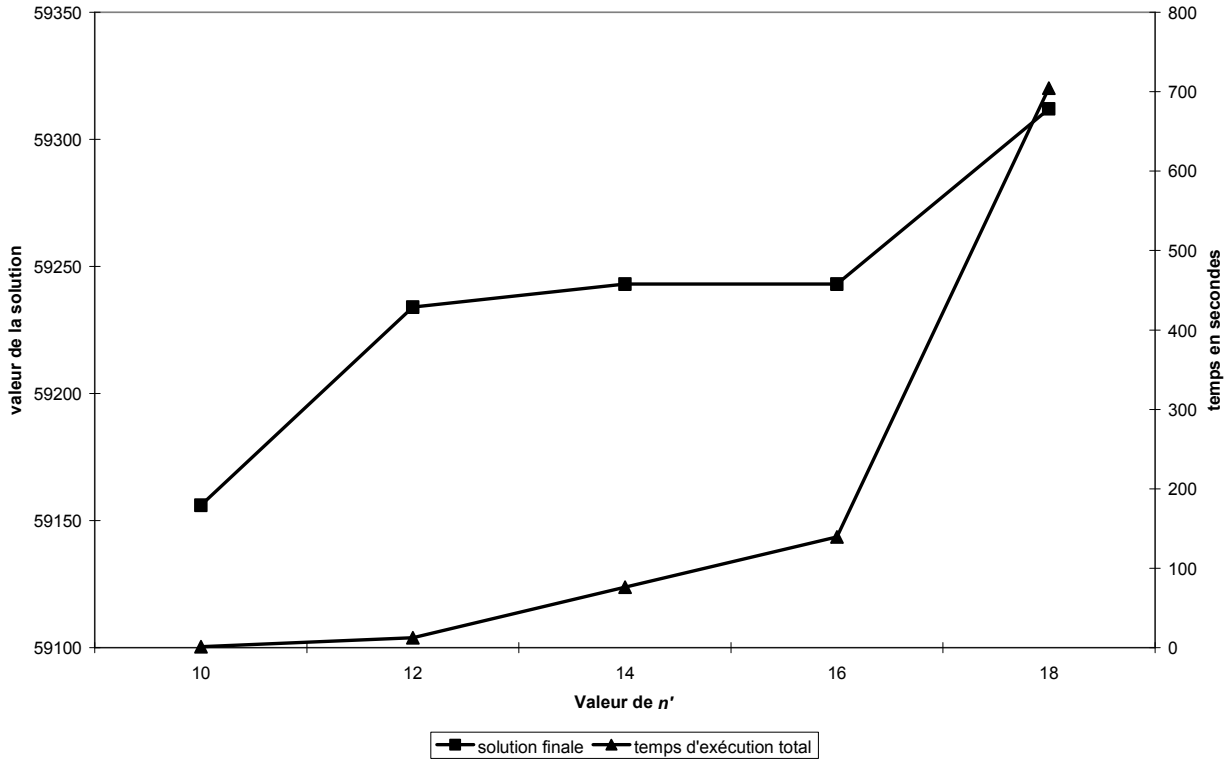

Figure 3.1 – Illustration de l'impact de *n'* sur le comportement de l'*IG*.

La valeur optimale de ce problème est de 59312. Elle est obtenue lorsque nous fixons *n'* à 18. Cependant le temps d'exécution total de l'approche augmente fortement avec cette valeur. L'augmentation importante entre les valeurs 16 et 18 s'explique également par le fait que l'algorithme tabou est appliqué une fois de plus dans le second cas. Pour les valeurs de *n'* égales à 10 et 12 le temps total est vraiment très bref (1 et 12 secondes). Lorsque la valeur de *n'* est égale à 14 ou 16 nous obtenons une solution ayant un coût de 59243 pour un temps

d'exécution correct. Notons aussi que le fait d'augmenter la valeur de *n'* n'assure en aucun cas une amélioration de la solution finale obtenue par le processus d'intensification globale. Cette figure montre que le choix de la valeur de *n'* n'est pas simple. Notre calcul permet cependant en général d'obtenir un bon compromis entre la qualité de la solution et le temps d'exécution comme sur cet exemple.

3.5 Conclusions

Nous avons présenté dans ce chapitre une approche hybride utilisant à la fois la programmation dynamique et la recherche tabou pour résoudre des problèmes en nombres entiers. L'approche consiste dans un premier temps à définir une bipartition de l'ensemble des variables en deux sous-ensembles. La programmation dynamique est appliquée sur le premier sous-ensemble pour résoudre une famille de sous-problèmes dont seul le second membre change. Seule la phase de construction de la programmation dynamique est utilisée au cours de cette partie de l'algorithme. Des concepts de réduction sont également intégrés pour améliorer la programmation dynamique et essayer de réduire la taille du problème initial. Un algorithme de recherche tabou est ensuite exécuté sur le sous-ensemble de variables complémentaire de manière à améliorer la solution obtenue. Chaque solution partielle rencontrée pendant la recherche est complétée en faisant appel à la phase de reconstruction de la programmation dynamique. L'ensemble de la procédure est répété tant que la solution peut être améliorée et/ou qu'il est possible de fixer des variables à leur valeur optimale. Dans le contexte de la recherche tabou l'approche peut être vue comme un processus d'intensification globale puisque chaque mouvement induit la résolution exacte d'un problème réduit.

Nous avons validé cette approche sur le problème du sac-à-dos multidimensionnel. Nous avons montré l'efficacité de la méthode sur un ensemble d'instances existantes et générées aléatoirement, en comparaison avec un logiciel d'optimisation efficace et un algorithme de recherche tabou seul. Nous avons également discuté de l'influence de quelques paramètres de l'algorithme.

Le chapitre suivant est consacré à l'application d'une approche basée sur une autre métaheuristique. Nous utilisons la recherche dispersée comme élément central dans laquelle nous introduisons quelques concepts de la recherche tabou.

Chapitre 4

Recherche dispersée et population diverse : étude et application au problème du sac-à-dos multidimensionnel

Nous abordons dans ce chapitre une autre métaheuristique pour la résolution de problèmes d'optimisation combinatoire. La recherche dispersée (*scatter search* selon la terminologie anglaise) est une méthode qui peut être classifiée parmi les algorithmes évolutifs, au même titre que les algorithmes génétiques. Une population de solutions, souvent appelée population *référence*, est utilisée tout au long du processus pour conserver un ensemble de solutions élites et diverses. La taille de cette population est généralement assez limitée (20 à 40 individus). Certains concepts sont étroitement liés à ceux utilisés dans les algorithmes génétiques. Il existe cependant de réelles différences entre ces deux approches. Plusieurs algorithmes génétiques assez efficaces existent pour résoudre le problème du sac-à-dos multidimensionnel. Il n'existe cependant pas, à notre connaissance, de références récentes basées sur la recherche dispersée et présentant des résultats numériques pour la résolution du problème du sac-à-dos multidimensionnel. Un des avantages de la recherche dispersée et des algorithmes génétiques est aussi qu'ils génèrent plusieurs solutions de bonne qualité. Cela peut être intéressant dans différents contextes lorsqu'il est demandé de fournir plusieurs solutions élites.

Après une description générale de la recherche dispersée et de quelques applications existantes, nous étudions dans ce chapitre le comportement de différents générateurs de solutions diverses. Ces générateurs peuvent être utilisés pour obtenir une population initiale et/ou pour fournir un ensemble de nouvelles solutions au cours de la recherche. Nous comparons les résultats obtenus par plusieurs générateurs en termes de qualité de la meilleure solution et de la diversité de l'ensemble des solutions générées. Nous décrivons dans les

sections 4.3 et 4.4 différents moyens pour gérer la mise à jour de la population référence et les nouvelles solutions obtenues. Nous abordons dans la section 4.5 un ensemble d'expériences numériques menées sur le problème du sac-à-dos multidimensionnel. Cette section est divisible en deux grandes parties. Nous commençons par une étude expérimentale visant à montrer l'influence de différents paramètres de l'algorithme sur ses performances. Nous donnons ensuite les résultats finaux obtenus sur un ensemble d'instances existantes. Nous terminons ce chapitre par les conclusions sur ce travail.

4.1 Principes de la recherche dispersée et applications

La recherche dispersée est une métaheuristique proposée par Glover et dont les concepts de base ont été formulés dans les années 1960. Les fondements de la méthode dérivent de stratégies initialement proposées et décrites par Glover [Glo63, Glo65] pour combiner des règles de décisions et des contraintes. L'objectif de la procédure est de générer de nouvelles solutions à partir des solutions courantes de manière à obtenir de meilleures solutions. Le processus de recherche dispersée est organisé de telle sorte que :

- Il doit exploiter et soustraire de l'information lors de la combinaison des solutions. Cette information n'apparaît pas nécessairement dans les éléments originaux pris séparément.

- Il doit utiliser différents processus heuristiques pour évaluer les solutions combinées et générer de nouveaux éléments.

La forme originale de la recherche dispersée est fortement liée à la notion de contrainte surrogate [Glo77]. Elle peut être présentée sous la forme suivante en trois étapes.

Etape 1 : Générer un ensemble de solutions à l'aide de différentes heuristiques adaptées au problème à résoudre. Désigner un sous-ensemble de meilleures solutions comme étant les solutions références.

Etape 2 : Créer des nouveaux points en appliquant des opérateurs de combinaisons linéaires à des sous-ensembles de solutions références. Les combinaisons linéaires sont :

(a) Choisies pour produire des points dans et en dehors des régions convexes couvertes par les solutions références.

(b) Modifiées par des processus d'arrondi pour obtenir des solutions entières si cela n'est pas le cas (dans le cas des problèmes en nombres entiers).

Etape 3 : extraire un ensemble de meilleures solutions générées à l'étape 2 et les utiliser comme point de départ des heuristiques de l'étape 1. Répéter le processus jusqu'à avoir effectué un certain nombre d'itérations.

La recherche dispersée repose en résumé sur les trois points fondamentaux suivants :

- Il est possible de récupérer de l'information concernant les solutions optimales du problème à partir d'un ensemble de solutions élites diverses.

- Il est important de définir les méthodes heuristiques utilisées lors de la combinaison des solutions de manière à ce qu'elles puissent générer des solutions dans des régions de l'espace non représentées par les solutions initiales. Cela permet de garantir un seuil de diversité tout en assurant un niveau de qualité.

- Le fait de considérer plusieurs solutions (au moins deux mais plus de deux si possible) lors des combinaisons permet d'exploiter l'information contenue dans l'union de ces solutions.

Une autre métaheuristique appelée méthode des chemins reliants (*path relinking* en anglais) est également en relation avec la recherche dispersée. Glover propose ainsi en 1998 [Glo98] un modèle pour développer un algorithme de recherche dispersée et/ou de chemins reliants. Il repose sur la définition des cinq composants majeurs (Ci, i = 1 à 5) suivants.

C1) Un algorithme permettant de générer un ensemble de solutions diverses. L'objectif est d'obtenir une population de solutions présentant un éventail de caractéristiques aussi large que possible à partir d'une solution initiale. Les solutions produites peuvent être réalisables ou non.

C2) Un processus d'amélioration. Cette méthode vise à obtenir une ou plusieurs solutions meilleures que la solution initiale. Il doit aussi pouvoir gérer les solutions non réalisables obtenues avec le composant C1.

C3) Un algorithme pour la construction et la mise à jour de la population référence. Ce composant doit pouvoir construire et modifier l'ensemble des meilleures solutions rencontrées au cours de la recherche. Celui-ci contient généralement un nombre peu élevé de solutions (20 à 40). Le terme *meilleure* ne correspond pas à la seule valeur de l'objectif du problème mais intègre aussi la notion de diversité.

C4) Un composant permettant la génération de sous-ensembles de solutions de l'ensemble référence. Ces sous-ensembles sont utilisés dans le dernier processus pour la génération de nouvelles solutions.

C5) Un algorithme de combinaison de sous-ensembles qui produit des nouvelles solutions qui servent ensuite comme point de départ à l'application des composants C1 à C5.

Un modèle d'algorithme de recherche dispersée en deux étapes peut être décrit de la manière suivante. La première étape est une étape d'initialisation servant à générer une population référence initiale à l'algorithme. La phase de recherche dispersée est exécutée au cours de la seconde étape.

Etape 1 : *Phase initiale*

Etape 1.1 : créer une ou plusieurs solutions initiales (réalisables ou non) du problème.

Etape 1.2 : générer un ensemble de solutions à l'aide du composant (C1) à partir de la/les solution(s) précédente(s).

Etape 1.3 : appliquer le composant (C2) à chaque solution obtenue en 1.2 pour générer une ou plusieurs meilleures solutions. Construire, puis maintenir à jour, la population référence au fur et à mesure à l'aide de (C3).

Etape 1.4 : Répéter les étapes 1.2 et 1.3 jusqu'à avoir atteint un nombre souhaité de solutions candidates pour la population référence.

Etape 2 : *Recherche dispersée*

Etape 2.1 : générer des sous-ensembles de solutions de la population référence avec (C4).

Etape 2.2 : pour chaque sous-ensemble construit, utiliser le composant (C5) pour obtenir une ou plusieurs solutions combinées.

Etape 2.3 : pour chaque solution générée précédemment appliquer (C2) et (C3).

Etape 2.4 : répéter les étapes 2.1 à 2.3 jusqu'à avoir effectué un nombre d'itérations fixé.

Les algorithmes de recherche dispersée que nous proposons respectent tous ce schéma. Nous confondons dans la suite population référence, ensemble référence et solutions références. La recherche dispersée a déjà été appliquée à différents problèmes d'optimisation combinatoire et les résultats obtenus sont généralement assez encourageants. Da Silva, Clímaco et Figueira ont récemment proposé deux applications de la recherche dispersée au problème du sac-à-dos bi-critères (ou bi-objectifs) [SCF06a] et au problème du sac-à-dos multidimensionnel bi-critères [SCF06b]. Dans la première approche, les auteurs utilisent deux propriétés pour générer un ensemble de solutions non dominées. La première est une propriété du problème du sac-à-dos monocritère. Une solution optimale du problème *SAD* diffère d'une

solution optimale de la relaxation en continu en un petit nombre de variables. Lorsque celles-ci sont considérées selon le rapport habituel c_j/a_j les variables qui diffèrent sont proches de la seule variable fractionnaire de cette solution. La deuxième propriété est proposée par les auteurs suite à différentes mesures expérimentales. Ces mesures montrent qu'une solution efficace du problème *SAD-BO* diffère généralement d'une autre solution efficace en un petit nombre de variables. Ces deux propriétés permettent de générer un ensemble de solutions de bonne qualité dans le cas de la première, et d'appliquer une méthode de recherche locale sur un espace de recherche restreint pour la seconde. Une des particularités de l'algorithme de recherche dispersée mis en œuvre se situe au niveau de la gestion de la population référence. Les auteurs utilisent la seconde propriété pour définir des groupes de solutions représentés par une seule d'entre elles au sein de la population, chaque groupe étant composé de solutions très proches. La population référence comprend au plus 20 groupes de solutions. Les sous-ensembles de solutions sont formés de deux solutions consécutives dans la population. Les auteurs utilisent une liste tabou pour éviter de générer plusieurs fois un même sous-ensemble. Dans le second article, les auteurs proposent une extension du travail précédent au cas multidimensionnel. Ils utilisent la relaxation surrogate pour transformer le problème initial en un problème de sac-à-dos bi-objectifs. Le multiplicateur utilisé est déterminé à l'aide d'un algorithme de sous-gradient. Les méthodes de diversification, amélioration des solutions et combinaison sont adaptées à cette nouvelle approche mais respectent le même principe que précédemment. Les résultats obtenus montrent que la recherche dispersée permet l'obtention de nombreuses solutions élites et peut rivaliser avec d'autres métaheuristiques sur ces deux problèmes bi-objectifs.

Marti, Lourenço et Laguna [MLL00] ont proposé un algorithme pour l'affectation des surveillants aux examens pour l'université de Barcelone. Le problème est tout d'abord formulé comme une extension du problème d'affectation généralisé avec deux objectifs : maximiser les préférences des surveillants et minimiser le nombre de surveillants nécessaires (le problème est ensuite ramené à un problème mono-objectif). Une des particularités de l'approche réside dans la définition du générateur de solutions diverses. Celui-ci prend en compte la fréquence des solutions déjà présentes dans la population. Lorsqu'une solution est générée plusieurs fois, la valeur de la préférence d'un surveillant pour une matière est modifiée pour forcer le générateur à obtenir d'autres solutions diverses. Quatre types de sous-ensembles sont générés selon la description de Glover [Glo98] sur laquelle nous revenons dans la section 4.4. La méthode de combinaison des solutions repose sur l'intégration d'un

vote qui consiste à déterminer pour chaque sous-ensemble quel est le surveillant le plus « intéressant » à affecter pour chaque examen.

D'autres approches combinent à la fois des éléments de la recherche dispersée et de la recherche tabou. Ainsi, Cung et al. [CMMT96] ont proposé un algorithme pour résoudre le problème d'affectation quadratique. L'algorithme repose à différents niveaux sur l'aléatoire, en particulier pour le choix des solutions à combiner. La solution générée est une combinaison linéaire des solutions sélectionnées. La recherche tabou est intégrée dans le processus de manière à intégrer des phases de diversification et d'intensification. La diversification est obtenue en mémorisant et en utilisant la fréquence des solutions, alors que des phases d'intensification sont effectuées en appliquant un algorithme tabou sur chaque nouvelle solution réalisable obtenue.

Greistorfer [Gre03] a proposé un algorithme tabou utilisant une population de solutions pour la résolution d'un problème particulier de routage : le problème du postier chinois avec capacité. L'intégration de la recherche dispersée consiste à générer aléatoirement et à l'aide d'une heuristique gloutonne une population initiale diverse. La recherche tabou est lancée sur la meilleure solution. A chaque fois qu'une bonne solution est rencontrée, elle peut être ajoutée dans la population (si elle fait partie des meilleures ou si elle est diverse). Finalement la recherche tabou est relancée sur la nouvelle solution obtenue.

Terminons cette revue de la littérature en mentionnant le livre de Laguna et Marti [LM03] dans lequel les auteurs donnent de nombreux détails sur l'implémentation de la recherche dispersée ainsi que plusieurs applications.

4.2 Génération d'une population diverse

Nous abordons dans cette section l'étape 1 du modèle d'algorithme de recherche dispersée présenté dans la section 4.1 précédente. Cette étape consiste à générer une population initiale diverse. Nous commençons cette section par la présentation du concept de diversité d'une population. Nous continuons avec la description de quelques générateurs existants que nous avons implémentés et comparés selon deux critères : la qualité de la meilleure solution et la diversité de la population. Nous terminons par la présentation d'un nouveau générateur généralement plus coûteux mais qui permet d'obtenir une population initiale de bonne qualité.

4.2.1 Diversité

La gestion des solutions diverses consiste à assurer un certain degré de diversité au sein de la population de solutions, et elle repose sur une mesure. Celle-ci est définie à partir d'une (ou plusieurs) mesure(s) entre deux solutions et entre deux sous-ensembles de solutions. Formellement, une mesure $d(P)$ associée à une population de solutions P est définie à partir d'une mesure d définie sur P^2 et de deux extensions f et g définies sur P^p, où p est la taille de P. Etant donnée une mesure d qui associe à chaque couple de solutions (x,y) une valeur $d(x,y)$, l'extension $d(x,P)$ entre une solution x et un sous-ensemble de solutions P est définie par

$$d(x,P) = f\{d(x,y) : y \in P\}.$$

où f est une fonction définie sur P^p. La mesure $d(P)$ d'un sous-ensemble de solutions P est définie par

$$d(P) = g\{d(x,P) : x \in P\}.$$

où g est une fonction définie sur P^p. Dans la littérature, la mesure d est le plus souvent choisie comme étant une distance qui vérifie les propriétés suivantes :

D1) Positivité : Pour tout couple (x,y) on a $d(x,y) \geq 0$.

D2) Séparation : $d(x,y) = 0$ implique $x = y$.

D2) Symétrie : Pour tout couple (x,y) on a $d(x,y) = d(y,x)$.

D3) Inégalité triangulaire : $d(x,z) \leq d(x,y) + d(y,z)$.

Commençons par la définition d'une mesure entre deux solutions. Dans le cas des problèmes en variables binaires, une distance couramment utilisée est la distance de Hamming. Elle est définie entre deux solutions x et y comme suit

$$d_1(x, y) = \sum_{j \in N} |x_j - y_j|.$$

Cette mesure prend bien en compte l'intégrité des variables, mais un de ses inconvénients est qu'elle ne prend pas en compte les données du problème. Etant donné l'aspect heuristique de la recherche dispersée, d'autres mesures (qui ne sont pas forcément des distances) peuvent être considérées. La fonction d_2 mesure par exemple l'écart entre deux solutions par rapport à l'objectif du problème

$$d_2(x, y) = |cx - cy|.$$

La mesure d_2 ne permet pas forcément de mesurer l'écart entre deux solutions car elles peuvent avoir des coûts très proches et être en fait assez éloignées l'une de l'autre. Pour illustrer ce principe considérons l'exemple suivant. Soit un problème d'optimisation dont

l'objectif est le suivant : max $5x_1 + 3x_2 + 2x_3 + 2x_4 + x_5$. Les deux solutions x_1 = (1 0 0 0 0) et x_2 = (0 1 0 1 0) ont la même valeur 5, et $d_2(x_1, x_2) = 0$. Les deux solutions sont pourtant très différentes. Sur cet exemple la distance d_1 semble plus adaptée puisque $d_1(x_1, x_2) = 3$. La fonction d_2 n'est pas une distance car elle ne vérifie pas la propriété de symétrie qui stipule que seules des solutions identiques ont une distance nulle entre elles.

Il est également possible d'utiliser des mesures qui prennent en compte la réalisabilité des solutions. La fonction d_3 mesure par exemple l'écart de la consommation des ressources entre deux solutions

$$d_3(x, y) = ||Ax - Ay||.$$

Nous pouvons aussi définir des mesures en combinant d'autres mesures. Par exemple, une nouvelle mesure d_4 entre les solutions x et y peut être définie comme une combinaison linéaire des mesures précédentes de la manière suivante

$$d_4(x, y) = \alpha \times d_1(x, y) + \beta \times d_2(x, y) + \gamma \times d_3(x, y)$$

avec α, β et γ des réels positifs normalisés. Dans toute la suite de ce chapitre nous utilisons cependant la distance d_1 dans le but de minimiser le paramétrage de l'algorithme final.

En ce qui concerne la mesure associée à une population de solutions, plusieurs solutions sont à nouveau envisageables. Les extensions $d(x,P)$ les plus souvent utilisées sont les fonctions suivantes

- $d(x,P) = \text{Max}\{d(x,y) : y \in P\}$.
- $d(x,P) = \text{Min}\{d(x,y) : y \in P\}$.
- $d(x,P) = \text{Somme}\{d(x,y) : y \in P\}$.

Les extensions $d(P)$ les plus souvent utilisées sont les fonctions suivantes

- $d(P) = \text{Max}\{d(x,P) : x \in P\}$.
- $d(P) = \text{Min}\{d(x,P) : x \in P\}$.
- $d(P) = \text{Somme}\{d(x,P) : x \in P\}$.

Plusieurs combinaisons de ces extensions peuvent être considérées. Nous en avons principalement retenu deux. Soit la mesure, notée $d^1(P)$ pour une population P, définie par

$$d^1(P) = \text{Max}\{\text{Min}\{d(x,y) : y \in P\} : x \in P\}.$$

Dans ce cas nous calculons pour chaque solution de la population la plus petite mesure par rapport à toutes les autres solutions. La diversité de la population correspond à la plus grande de ces plus petites mesures. D'un point de vue géométrique et dans le cas de la distance d_1, cette mesure peut être traduite comme étant le plus grand rayon des cercles

centrés en une solution de la population, passant par une autre solution (au moins), et n'en contenant aucune à l'intérieur. La figure 4.1 illustre ce principe sur un ensemble de quatre solutions.

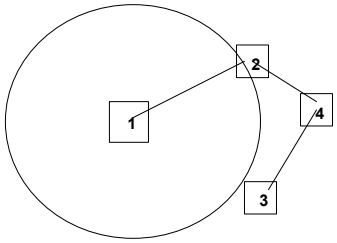

Figure 4.1 – Illustration de la plus grande des plus petites distances.

Dans la figure 4.1 chaque solution de 1 à 4 est représentée par un carré, et est reliée à la solution la plus proche selon la mesure d_1 par un trait. Le rayon du cercle détermine la diversité de la population (la valeur $d^1(P)$). Lors de la génération de la population diverse, nous commençons par ajouter les solutions sans critère particulier (excepté le fait que les doublons sont interdits). Une fois que le nombre de solutions souhaitées est atteint nous déterminons $d^1(P)$. Lorsqu'une nouvelle solution est candidate nous calculons l'ensemble des distances par rapport aux solutions déjà présentes selon la mesure d_1, et nous conservons la plus petite de ces distances. Si celle-ci est supérieure à $d^1(P)$ la nouvelle solution remplace la solution la *moins diverse*, c'est-à-dire celle qui a la plus petite des plus petites mesures. Sur l'exemple de la figure 4.1, il s'agit de la solution 2 ou de la solution 4 puisque $d_1(2, 4) < d_1(3, 4)$. Le choix entre les deux solutions se fait selon la seconde plus petite distance par rapport aux autres solutions de l'ensemble. Dans la figure 4.1 nous choisissons la solution 4 car ici nous avons clairement $d_1(4, 3) < d_1(2, 3)$ et $d_1(4, 3) < d_1(2, 1)$. En cas de nouvelle égalité (ce cas peut arriver facilement en pratique), nous éliminons en priorité la solution de l'ensemble ayant la plus petite valeur associée à l'objectif du problème.

Notons qu'une solution candidate peut aussi être insérée dans la population si sa plus petite distance est supérieure à la plus petite des plus petites distances. L'insertion de la solution dans ce cas n'est faite que si la valeur $d^1(P)$ ne diminue pas. Ajoutons que d'un point de vue algorithmique nous ne mesurons pas la distance entre les candidats et la solution la moins diverse de la population puisque celle-ci ne restera pas en cas d'ajout. Ce genre de stratégie d'ajout d'une solution diverse a par exemple été utilisé par Marti, Lourenço et Laguna [MLL00].

Il est également possible d'utiliser une approche duale à la précédente. Nous conservons pour chaque solution la plus grande distance par rapport à toutes les autres de l'ensemble, et la diversité de la population est égale à la plus petite de ces plus grandes distances. Géométriquement, cette mesure correspond au plus petit cercle centré en un point de l'ensemble et qui contient toutes les autres solutions. La figure 4.2 illustre cette approche.

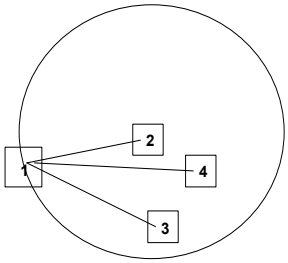

Figure 4.2 – Illustration de la plus petite des plus grandes distances.

Formellement cette distance est définie comme suit

$$d^2(P) = \text{Min}\{\text{Max}\{d(x,y) : y \in P\} : x \in P\}.$$

Nous utilisons la première approche dans la suite de ce chapitre. Nous présentons dans la partie suivante plusieurs générateurs de solutions diverses que nous avons implémentés. Nous comparons les performances de ces générateurs sur un ensemble d'instances existantes de problèmes de sac-à-dos multidimensionnel.

4.2.2 Générateurs de solutions diverses

4.2.2.1 Générateur aléatoire

Le générateur de solutions aléatoires que nous avons utilisé est applicable à tous les problèmes en variables 0-1. La seule donnée du problème nécessaire est le nombre de variables n. Nous proposons également de fournir en paramètre le nombre de solutions distinctes souhaitées p, ainsi qu'un intervalle correspondant au nombre de variables de décisions fixées à 1 au minimum et au maximum dans chaque solution [*Mini*, *Maxi*].

Le principe de ce générateur est assez simple et peut être résumé comme suit. Tant que le nombre de solutions générées n'a pas atteint la valeur souhaitée, nous tirons une valeur $n1$ au hasard dans l'intervalle [*Mini*, *Maxi*]. Nous tirons ensuite successivement des indices de variables compris entre 1 et n jusqu'à avoir $n1$ variables fixées à 1. Si la solution a déjà été générée alors elle n'est pas choisie, sinon elle est ajoutée à la population. L'algorithme 4.1 présente cette méthode.

Algorithme 4.1 Générateur aléatoire

Entrées : le nombre de variables du problème n; le nombre de solutions à générer p; l'intervalle [*Mini*, *Maxi*].
Sortie : une population P.

```
 1 : P = ∅; k = 0;
 2 : tant que k < p faire
 3 :    n' = n; n1 = aléa(Mini, Maxi);
 4 :    pour j = 1 à n' faire J[j] = j;
 5 :    x = (0 0 … 0);
 6 :    pour j = 1 à n1 faire
 7 :       Permuter(J[aléa(1,n')],J[n']);
 8 :      n' = n' – 1;
 9 :    fin pour
10 :    pour j = 1 à n1 faire x[J[j]] = 0;
11 :    pour j = n1+1 à n faire x[J[j]] = 1;
12 :    si x ∉ P alors P = P ∪ {x} ; k = k + 1;
13 : fin tant que
14 : retourner P
```

Dans cet algorithme, la population générée est représentée par P. La fonction notée *aléa* retourne un entier compris entre les deux valeurs passées en paramètre. La fonction *Permuter* permute les deux valeurs passées en paramètre dans le tableau J de dimension n. Celui-ci est utilisé pour éviter de tirer plusieurs fois une même variable au cours de la génération d'une solution. La variable n' correspond au nombre de variables non encore tirée au sort lors de la génération de la solution courante.

L'avantage de ce type de générateur est qu'il est peu coûteux en temps d'exécution. Il est facilement adaptable à de nombreux problèmes. L'inconvénient principal est qu'il ne prend pas en compte les données du problème, ce qui fait que les solutions obtenues peuvent être réalisables, non réalisables et très éloignées des solutions optimales. Notons qu'il n'y a pas non plus de garantie sur la diversité de la population obtenue.

4.2.2.2 Générateur séquentiel

Glover a proposé dans [Glo98] un générateur de solutions diverses pour les problèmes en variables 0-1. Ce générateur repose sur deux paramètres : une solution initiale du problème (réalisable ou non), et un entier qui permet de limiter le nombre de solutions générées. Nous notons cet entier h^*. De la même manière que l'algorithme précédent, ce générateur ne prend pas en compte les autres données du problème. Il peut donc générer des solutions réalisables comme des solutions non réalisables. Glover ne précise pas explicitement le nombre de solutions obtenues avec cet algorithme mais donne une approximation : $h^* \times (h^* + 1)$. Nous

proposons plusieurs formules permettant de déterminer exactement le nombre de solutions générées. Nous mettons également en évidence un phénomène de répétition susceptible de se produire lors du processus.

A chaque itération h de l'algorithme deux solutions complémentaires sont générées. Le premier type de solution, noté x', est déterminé comme suit.

$$x'_1 = 1 - x_1$$

$$x'_{1+jh} = 1 - x_{1+jh}, \text{ pour } j = 1, 2, \ldots, j^*$$

avec x la solution initiale fournie au générateur, et j^* la partie entière inférieure de n / h (i.e. $j^* = \lfloor n / h \rfloor$). Les autres variables de décision de la solution x' sont fixées à 0. La seconde solution, x'', est égale à la solution complémentaire de x'.

Il est possible de stopper le processus lorsqu'un certain nombre de solutions a été généré ou lorsque l'ensemble des solutions associées au paramètre h^* a été produit. Il est aussi possible de l'appliquer jusqu'à avoir obtenu un certain nombre de solutions réalisables par exemple. L'algorithme 4.2 présente le fonctionnement de ce générateur que nous appelons générateur séquentiel dans la suite. Les paramètres d'entrée de l'algorithme sont le nombre de variables du problème, n, une solution initiale, x et l'entier h^*. La population P est récupérée en sortie de l'algorithme.

Algorithme 4.2 Générateur séquentiel

Entrées : le nombre de variables du problème n; la solution initiale x; h^*.
Sortie : la population P.

```
 1 : P = ∅; k = 1;
 2 : pour h allant de 1 à h* faire
 3 :    si h < 3 alors i* = 1;  sinon i* = h;
 4 :    pour i allant de 1 à i* faire
 5 :       pour j allant de 1 à ⌊(n – i) / h⌋ faire x'_{i+jh} = 1 – x_{i+jh};
 6 :       P[k] = x'; k = k + 1;
 7 :       si h > 1 alors x'' = e – x'; P[k] = x''; k = k + 1;
 8 :   fin pour
 9 : fin pour
10 : retourner P
```

Glover précise que la ligne 3 permet d'éviter de générer des solutions déjà produites. La ligne 7 évite de générer la solution initiale x. Pour définir le nombre exact de solutions générées par cet algorithme, nous notons le nombre de solutions distinctes par $D_{n,h}$ et le nombre de répétitions par $R_{n,h}$ (avec h la valeur de h^* dans un soucis de simplification). Nous utilisons également le paramètre $p = \lfloor n / 2 \rfloor$ en remarquant que n est entier, ce qui implique

$n = 2 \times p$ si n est pair.

$n = 2 \times p + 1$ si n est impair.

Le nombre de solutions distinctes générées par l'algorithme 4.2 est donné par les formules suivantes.

$D_{n,1} = 1$

$D_{n,2} = 3$

$D_{n,h} = h(h+1) - 3$, pour $h = 3, \ldots, p+1$

$D_{n,h} = (p+1)(p+2) - 3 + (n-p)(n-p+1) - (n-h+1)(n-h+2)$, pour $h = p+2, \ldots, n$

Le nombre de répétitions est lui déterminé par les formules suivantes.

$R_{n,h} = 0$, pour $h = 1, 2, \ldots, p + 1$

$R_{n,h} = (h-p)(h-p-1)$, si n est pair et $h = p+2, \ldots, n$

$R_{n,h} = (h-p)(h-p-2) + 1$, si n est impair et $h = p+2, \ldots, n$

Ces formules ont été obtenues à partir d'un ensemble de mesures expérimentales en exécutant le générateur pour différentes valeurs de n et de h^*. Nous avons appliqué l'algorithme sur un ensemble d'instances de tailles différentes. Les formules ont été déterminées à partir des résultats et ont été vérifiées sur l'ensemble des instances.

Nous pouvons remarquer avec l'expression de $R_{n,h}$ qu'il n'y a pas de phénomène de répétition jusqu'à la valeur $h^* = n / 2$. Cette valeur peut ainsi être utilisée comme paramètre de l'algorithme. Glover précise que la valeur $n / 5$ est recommandée (l'algorithme peut fournir des solutions jusque $h^* = n - 1$). Pour illustrer les formule précédentes, nous donnons dans le tableau 4.1 quelques valeurs obtenues en fonction du nombre de variables du problème et de la valeur de h^*.

h^*	n	10	15	20	25	50	75	100	150	200
n-1	$D_{n,h}$	87	207	377	597	2447	5547	9897	22347	39797
	$R_{n,h}$	12	36	72	121	552	1296	2352	5402	9702
n-2	$D_{n,h}$	69	179	339	549	2349	5399	9699	22049	39399
	$R_{n,h}$	6	25	56	100	506	1225	2256	5256	9506
$\lfloor n/2 \rfloor$	$D_{n,h}$	27	53	107	153	647	1403	2547	5697	10097
	$R_{n,h}$	0	0	0	0	0	0	0	0	0
$\lfloor n/2 \rfloor$-1	$D_{n,h}$	17	39	87	129	597	1329	2447	5547	9897
	$R_{n,h}$	0	0	0	0	0	0	0	0	0

Tableau 4.1 – Nombre de solutions produites par le générateur séquentiel.

Le tableau 4.1 montre que le nombre de solutions obtenues augmente très rapidement avec n et h^*. Pour les instances de petites tailles ($n \leq 25$), il semble qu'une valeur de h^* assez grande soit plus intéressante car elle permet d'obtenir un nombre de solutions distinctes nettement supérieur à la valeur de n, sans toutefois devenir trop important. Par contre lorsque la taille augmente, une valeur inférieure à $n/2$ paraît suffisante pour générer un nombre

conséquent de solutions distinctes (et cela assure également le fait de ne pas avoir de répétitions).

Ce générateur présente théoriquement un avantage par rapport au générateur aléatoire présenté précédemment au niveau de la diversité. En effet ici le processus de génération des solutions est guidé, alors qu'au cours du processus de génération aléatoire seul le hasard intervient.

4.2.2.3 Générateur dichotomique

Glover propose dans le même article [Glo98] un autre processus pour générer un ensemble de solutions diverses pour lequel il est facile de déterminer le nombre de solutions distinctes obtenues. Le but de l'algorithme est d'obtenir des solutions les plus éloignées possibles entre elles. Il utilise à nouveau comme paramètre le nombre de variables du problème, n. Il permet la génération de $2 \times (1 + \log n)$ solutions (notons que la solution initiale et sa solution complémentaire sont toutes deux considérées comme des solutions générées). Lorsque n est une puissance de deux, chaque solution obtenue maximise la distance minimale de Hamming par rapport à toutes les autres solutions (i.e. selon la formule (4.1)). Du point de vue de la diversité, ce générateur paraît donc plus intéressant que les précédents car il prend en compte la diversité de la population toute entière.

L'algorithme 4.3 décrit le fonctionnement de ce générateur. Nous l'appelons générateur dichotomique en référence à son principe qui consiste à partitionner l'ensemble N en plusieurs sous-ensembles puis à complémenter la solution initiale sur différentes unions de ces sous-ensembles. Le processus est répété jusqu'à n'avoir que des partitions comportant un élément. Soit x une solution binaire sur N et soit J un sous-ensemble de N, nous notons $\bar{x}(J)$ le complémentaire de x relatif à J, défini par

$$\bar{x}(J) = \begin{cases} 1 - x_j & \text{si } j \in J \\ x_j & \text{sinon} \end{cases}$$

Algorithme 4.3 Générateur dichotomique

Entrées : le nombre de variables du problème n; la solution initiale x.
Sortie : la population P.

1 : *Initialisation* : $P[1] = \bar{x}(\varnothing)$; $P[2] = \bar{x}(N)$; $k = 3$;
2 : *Partitionnement* : Soit $N = N' \cup N''$ avec $|N'| = |N''| \pm \varepsilon$ et $\varepsilon \in \{0,1\}$. Poser $P[k] = \bar{x}(N')$ et $P[k+1] = \bar{x}(N'')$; $k = k + 2$;
3 : *Construction* : Partitionner chaque sous-ensemble S de N en 2 sous-ensembles S' et S'' et poser $N' = \cup_N S'$; $N'' = \cup_N S''$; Poser $P[k] = \bar{x}(N')$ et $P[k+1] = \bar{x}(N'')$; $k = k + 2$;
4 : *Terminaison* : **si** $\exists\, S \in N$ tel que $|S| > 1$ **alors** aller à Construction; **sinon** retourner P.

Dans l'algorithme 4.3, l'ensemble *N* est partitionné en deux sous-ensembles *N'* et *N''* (d'une taille aussi proche que possible). La solution initiale est complémentée sur *N'* et sur *N''*, produisant deux nouvelles solutions. Le processus est répété en partitionnant à chaque itération chaque sous-ensemble précédent en deux nouveaux sous-ensembles. La solution initiale est complémentée sur l'union des sous-ensembles *N'* et sur celle des sous-ensembles *N''*. Le processus s'arrête lorsque la condition de la ligne 4 est vérifiée.

Nous présentons un exemple d'application de ce générateur pour un problème comportant 16 variables. Le générateur construit dans ce cas 8 solutions auxquelles s'ajoutent la solution initiale et son complémentaire. Nous occultons ces deux dernières solutions et ne présentons que les solutions *x*' générées par l'algorithme 4.3 dans un souci de clarté (les solutions *x*'' sont faciles à obtenir).

Exemple 4.1 :

1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0
1	**1**	**1**	**1**	0	0	0	0	**1**	**1**	**1**	**1**	0	0	0	0
1	**1**	0	0	**1**	**1**	0	0	**1**	**1**	0	0	**1**	**1**	0	0
1	0	**1**	0	**1**	0	**1**	0	**1**	0	**1**	0	**1**	0	**1**	0

Dans cet exemple les valeurs en gras correspondent à l'union *N'* des sous-ensembles *S'*, et les éléments soulignés à l'union *N''* des sous-ensembles *S''*.
La solution de la seconde ligne est par exemple obtenue en partitionnant chacun des deux sous-ensembles {1,2,…,8} et {9,10,…,16} de la première ligne. Le premier est partitionné pour donner les deux nouveaux sous-ensembles {1,2,3,4} (*S'*) et {9,10,11,12} (*S''*), alors que le second donne les deux sous-ensembles {5,6,7,8} (*S'*) et {13,14,15,16} (*S''*). Pour déterminer la solution nous complémentons la solution initiale sur l'union des sous-ensembles *S'*. Le choix du partitionnement est un élément important dans le fonctionnement du générateur. Nous avons choisi dans cet exemple de partitionner *N* en fixant *N'* = {1,2,…,8} et *N''* = {9,10,…,16}. Lors du second partitionnement nous divisons à nouveau les sous-ensembles en deux en considérant les indices dans l'ordre. Il s'agit de la version la plus simple et il est possible de modifier l'ordre des variables.

Nous avons testé les trois générateurs précédents sur 55 instances classiques de problèmes de *SADM* de petites tailles (*n* allant de 6 à 105 et *m* de 2 à 30). Ces instances sont disponibles sur Internet [Bea90, Sko02] et ont été générées lors de la publication de différents

articles. Pour obtenir suffisamment de solutions avec le générateur dichotomique, nous utilisons plusieurs partitions en modifiant l'ordre des variables. Nous générons pour cela la séquence *jp* (modulo *n*), avec *p* un nombre premier, et *j* un entier qui varie de 1 à *n*. Cette technique a été suggérée par Glover [Glo98] et permet de construire plusieurs populations avec la même solution initiale.

Nous avons utilisé deux types de solutions initiales pour les générateurs séquentiel (*Séq*) et dichotomique (*Dicho*). Nous utilisons dans le premier cas une solution initiale nulle et dans le second cas la solution retournée par une heuristique basée sur la programmation linéaire (*HPL*) présentée dans le chapitre 5 (section 5.1). Pour le générateur aléatoire (*Aléa*), nous lançons 20 exécutions différentes et les résultats sont une moyenne sur les 20 exécutions. Le nombre de variables égales à 1 dans chaque solution est compris entre 15 et 85%. Pour le générateur séquentiel nous utilisons la valeur $n - 1$ pour le paramètre h^*.

Comme les trois générateurs fournissent des solutions réalisables et non réalisables, nous effectuons deux tests consécutifs. Nous commençons par appliquer chaque générateur pour produire $5 \times n$ solutions distinctes et nous récupérons les 20 solutions les plus diverses parmi celles-ci (réalisables ou non). Nous appliquons ensuite sur chaque solution une phase d'amélioration à l'aide d'un algorithme glouton pour ne conserver que les 20 solutions réalisables les plus diverses obtenues. Nous gardons également dans les deux cas la meilleure solution réalisable.

Pour pouvoir faire une analyse globale des résultats des trois générateurs, nous utilisons une mesure de la *performance* pour chaque population P^i selon les deux critères évalués : la qualité des solutions et la diversité de la population. Nous calculons pour cela les deux valeurs Δd^i et Δc^i pour chaque population P^i et pour chaque problème de la manière suivante.

$$\Delta d^i = \frac{d^i - d_{\min}}{d_{\max} - d_{\min}}, \text{ avec } d^i = \sum_{x \in P_i} d\left(x, P^i\right), \text{ et } d_{\min} = \min_i \left\{d^i\right\} \text{ et } d_{\max} = \max_i \left\{d^i\right\}.$$

d^i est la somme des plus petites distances entre les solutions de la population P^i dans notre cas.

La valeur Δc^i est obtenue de la même manière :

$$\Delta c^i = \frac{c^i - c_{\min}}{c_{\max} - c_{\min}}, \text{ avec } c^i = \sum_{x \in P_i} cx, \text{ et } c_{\min} = \min_i \left\{c^i\right\} \text{ et } c_{\max} = \max_i \left\{c^i\right\}.$$

Ce genre de mesure a par exemple été utilisé par Campos et al. [CGLM01]. Plus (resp. moins) une population P^i est diverse par rapport aux autres et plus la valeur de Δd^i sera proche de 1 (resp. 0).

Nous donnons dans le tableau 4.2 la performance moyenne en terme de diversité pour la population initiale lorsque la solution initiale est nulle (d_0) et lorsque la solution initiale est fournie par l'heuristique *HPL* (d_{HPL}) pour chacun des 3 générateurs. Les colonnes d_0A et $d_{HPL}A$ donnent les performances moyennes en terme de diversité pour la population finale (après amélioration) selon la solution initiale utilisée.

Générateur	d_0	d_{HPL}	d_0A	$d_{HPL}A$
Séq	0.00	0.00	0.15	0.08
Dicho	1.00	1.00	0.69	0.97
Aléa	0.32	0.32	0.60	0.27

Tableau 4.2 – Performances des générateurs en terme de diversité.

Nous pouvons voir dans le tableau 4.2 que les deux solutions initiales utilisées conduisent au même résultat pour la diversité de la population initiale. Le générateur séquentiel obtient en fait la population initiale la plus diverse pour une seule instance. C'est quasiment toujours ce générateur qui obtient la plus petite distance entre deux solutions au sein de la population. Le générateur dichotomique domine très nettement les deux autres en terme de diversité. Le fait d'appliquer une phase d'amélioration diminue un peu cette domination lorsque la solution initiale est nulle, mais c'est toujours le générateur séquentiel qui est le moins performant sur ces instances.

Dans le tableau 4.3 nous présentons les résultats en terme de qualité de la population. Nous donnons les résultats après la phase d'amélioration car avant cette phase il peut y avoir des solutions non réalisables dans les populations. La colonne c_0A précise la qualité moyenne finale de la population lorsque la solution initiale est nulle, alors que la colonne $c_{HPL}A$ donne celle avec la solution obtenue par *HPL*. Nous indiquons finalement dans les colonnes $C+D_0$ et $C+D_{HPL}$ la performance moyenne générale de chaque générateur en fonction de la solution utilisée selon les deux critères.

Générateur	c_0A	$c_{HPL}A$	$C+D_0$	$C+D_{HPL}$
Séq	0.72	0.55	0.87	0.63
Dicho	0.34	0.41	1.03	1.38
Aléa	0.44	0.54	1.04	0.81

Tableau 4.3 – Performances des générateurs en qualité et performance globale.

Nous pouvons voir dans le tableau 4.3 que le générateur séquentiel domine les deux autres en terme de qualité lorsque nous utilisons une solution nulle. La différence entre les trois générateurs est nettement plus faible avec une solution non nulle. Les deux dernières colonnes montrent que le générateur séquentiel est en moyenne dominé par les deux autres. Cela s'explique par les résultats du tableau 4.2 précédent. Remarquons aussi la bonne performance moyenne du générateur dichotomique, en particulier lorsque la solution initiale est non nulle.

Ces résultats semblent montrer que l'utilisation du générateur dichotomique est plus intéressante, en particulier au sein de l'algorithme de recherche dispersée puisque nous pouvons facilement utiliser une solution initiale non nulle. Les résultats obtenus par le générateur séquentiel sont un peu décevants, surtout par rapport au générateur aléatoire. Ils peuvent en partie s'expliquer par les 20 exécutions du générateur aléatoire qui permettent d'obtenir sur ces instances de meilleurs résultats moyens.

4.2.3 Générateur basé sur la programmation linéaire

Nous proposons un nouveau générateur de solutions diverses. Celui-ci se base sur l'heuristique *HPL* dont le principe est de résoudre la relaxation en continu du problème puis un problème réduit obtenu en fixant l'ensemble des variables entières d'une solution optimale de cette relaxation (chapitre 5, section 5.1). L'idée du générateur est d'appliquer une première fois l'heuristique *HPL* sur le problème initial, puis de réitérer le processus sur n problèmes réduits associés aux n variables du problème. La définition d'un problème réduit est donnée dans le chapitre 3 (section 3.1.3). Rappelons qu'il est obtenu à partir d'une solution du problème en fixant une partie des variables à leur valeur dans cette solution.

Le principe du générateur est alors le suivant. La première application de l'heuristique *HPL* nous permet de récupérer une solution réalisable x^0 et une solution optimale de la relaxation en continu $\bar{x}$ pour le problème *SADM*. A partir de la solution $\bar{x}$, nous générons les n problèmes réduits $SADM(e\text{-}x^0, \{j\})$ associés à chaque variable de décision x_j. Il est alors possible d'appliquer l'heuristique *HPL* sur chacun des problèmes obtenus pour calculer une nouvelle solution réalisable du problème. Notons qu'il est possible qu'une solution soit générée plusieurs fois, auquel cas elle n'est conservée qu'une seule fois dans la population. Ajoutons que ce processus peut générer des solutions non réalisables pour certains problèmes d'optimisation (ce n'est pas le cas pour le problème *SADM*). Dans ce cas il est possible d'appliquer un opérateur de réparation qui transforme cette solution en solution réalisable.

Nous appliquons également la propriété classique de fixation présentée dans le chapitre 3 (propriété 3.1 section 3.1.3) qui repose sur les problèmes réduits et une solution réalisable. Cela nous permet de réduire la taille du problème initial avant l'application de l'algorithme de recherche dispersée dans certains cas.

Nous décrivons ce générateur dans l'algorithme 4.4. La variable F désigne l'ensemble des variables fixées définitivement à leur valeur optimale par l'algorithme. Nous notons $HPL(SADM,x,\bar{x})$ l'application de l'algorithme *HPL* qui génère une solution réalisable x et une solution optimale de la relaxation en continu $\bar{x}$ du problème *SADM*. La ligne 5 correspond à l'application de la propriété de fixation.

Algorithme 4.4 Générateur de solutions diverses basé sur la programmation linéaire

Entrée : le problème *SADM*.
Sorties : la population P; l'ensemble des variables fixées F.

1 : $P = \emptyset$; $F = \emptyset$;
2 : $HPL(SADM,x^0,\bar{x})$;
3 : **pour** j allant de 1 à n **faire**
4 : $HPL(SADM(e\text{-}x^0,\{j\}),x,\bar{x})$;
5 : **si** $c\bar{x} \leq cx^0$ **alors** $F = F \cup \{j\}$; $N = N - \{j\}$; **fin si**
6 : **si** $x \notin P$ **alors** $P = P + \{x\}$; **fin si**
7 : **fin pour**
8 : retourner P

Cet algorithme permet de produire dans la majorité des cas une population initiale comportant un ensemble de solutions réalisables proches des meilleures solutions connues pour les instances de problèmes de sac-à-dos multidimensionnel que nous avons traitées. Le saut moyen des solutions par rapport aux meilleures solutions connues est ainsi de l'ordre de 0.2% sur l'ensemble de 270 instances du problème *SADM* présenté dans le chapitre 3 (section 3.4). La fixation moyenne est elle de l'ordre de 13.5%. Nous proposons plusieurs extensions de cet algorithme pour en améliorer les performances. Une des motivations est qu'il est possible d'augmenter le pourcentage de variables fixées à leur valeur optimale et ainsi de pouvoir appliquer l'algorithme de recherche dispersée sur de plus petits problèmes. Nous appliquons pour cela une phase d'amélioration (à l'aide d'un algorithme de recherche locale) sur chaque solution générée. Cela nous permet dans certains cas d'obtenir une meilleure solution plus rapidement et ainsi de fixer des variables plus vite. Une autre idée consiste à relancer l'algorithme tant que la meilleure solution est améliorée et/ou qu'il est possible de fixer au moins une variable. Nous utilisons alors la meilleure solution courante pour générer

les problèmes réduits. Le nombre total de problèmes réduits à résoudre reste dans ce cas borné comme le montre la propriété 4.1 suivante.

Propriété 4.1 : Soit x^{0k} la meilleure solution réalisable obtenue lors des k-1 premières itérations de l'algorithme (si $k = 1$ alors x^{0k} est la solution obtenue par l'application de *HPL* (ligne 2 de l'algorithme 4.4)). Si on a $x_j^{0k} = 1 - \bar{x}_j$ pour $j \in \{1,\ldots,n\}$ alors la valeur de la relaxation en continu du problème réduit $SADM(e\text{-}x^0, \{j\})$ est la même que celle du problème initial. Il est donc inutile de résoudre de manière optimale le problème réduit obtenu à partir de cette solution.

Notons que les heuristiques basées sur des relaxations présentées dans le chapitre 5 de ce mémoire peuvent également être utilisées comme générateur de solutions. Nous continuons ce chapitre en abordant les autres composants de l'algorithme de recherche dispersée. Lors de la génération de la population initiale, nous devons construire la population référence. La section suivante présente les concepts liés à cette construction et à la mise à jour de cette population.

4.3 Mise à jour de la population référence

Si nous suivons l'ordre des composants (C*i*) mentionné dans la section 4.1, nous devons discuter de l'amélioration des solutions. Nous utilisons dans nos algorithmes deux types d'amélioration selon que la solution est réalisable ou non. Les deux améliorations consistent à effectuer une phase de recherche locale basée sur un principe glouton. Lorsqu'une solution est non réalisable elle est projetée vers le domaine réalisable en retirant des objets un à un du sac. Nous stoppons le processus une fois que la solution est réalisable ou juste avant si nous souhaitons avoir une solution non réalisable proche de la frontière. Lorsqu'une solution est réalisable nous essayons de l'étendre vers la frontière en lui ajoutant, lorsque cela est possible, un ou plusieurs objets. Si la solution est déjà de bonne qualité nous appliquons l'heuristique *Complément* décrite dans le chapitre 3 (algorithme 3.4).

Le composant (C3) concerne la création et la mise à jour de la population référence qui contient les meilleures solutions et évolue au cours de la recherche. A chaque fois qu'une nouvelle solution ou un nouvel ensemble de solutions est généré, le processus de mise à jour de la population référence est appelé. Différents critères peuvent être utilisés pour choisir d'ajouter ou non une solution dans la population. Les deux critères classiques utilisés dans la

littérature sont le coût de la solution et la diversité de la solution. Si nous utilisons uniquement le premier critère alors il est clair que la population regroupe un ensemble de solutions élites, alors que dans le second cas elle regroupe un ensemble de solutions diverses. Il est généralement recommandé de diviser la population référence en deux sous-ensembles. Cela permet d'obtenir un compromis entre les deux critères.

Quelle que soit la stratégie adoptée, il est nécessaire d'interdire la duplication des solutions dans la population. La détection des duplications peut souvent se faire rapidement et facilement. Dans le cas des problèmes en variables 0-1, une solution est généralement représentée par un vecteur de *n* bits représentant les valeurs des *n* variables de décision. Une façon triviale de comparer deux solutions consiste à parcourir les deux vecteurs correspondants et à vérifier si chaque valeur est identique. L'inconvénient de cette approche est qu'elle nécessite *n* comparaisons dans le cas où les deux solutions sont identiques. D'autres informations peuvent être ajoutées pour représenter une solution et accélérer la comparaison. Nous utilisons dans notre approche le coût de la solution ainsi que l'ensemble des capacités résiduelles associées aux contraintes du problème. La détection de la duplication peut alors être simplifiée en considérant que deux solutions sont identiques si elles ont le même coût et les mêmes capacités résiduelles. Il se peut cependant que ce genre de méthode élimine des solutions distinctes ayant les mêmes caractéristiques. Cela est rarement le cas en pratique et se produit éventuellement sur les petites instances. L'exemple 4.2 illustre cette situation.

Exemple 4.2 : détection d'une duplication inexistante

Soit le problème *P*2 suivant

$$(P2)\quad \begin{cases} \max & 5x_1 + 4x_2 + 2x_3 + x_4 + x_5 & & \\ s.c. & 4x_1 + 2x_2 + x_3 + 2x_4 + x_5 & \leq & 4 \\ & 7x_1 + 4x_2 + 2x_3 + 3x_4 + 2x_5 & \leq & 8 \\ & x_j \in \{0,1\} \qquad j = 1,...,5 & & \end{cases}$$

Soient les deux solutions réalisables de ce problèmes x_1 = (1 0 0 0 0) et x_2 = (0 1 0 1 0). Ces deux solutions ont le même coût de valeur 5, et les mêmes capacités résiduelles, 0 et 1. Elles sont pourtant bien distinctes.

Cette technique diminue éventuellement la complexité temporelle de l'algorithme mais elle augmente la complexité spatiale puisque nous conservons *m* + 1 informations supplémentaires pour chaque solution. Une manière de limiter cette augmentation consiste à

ne conserver en mémoire que les indices des variables égales à 1 dans la solution (car elles sont généralement moins nombreuses que les variables égales à 0 pour le problème *SADM*).

4.4 Génération de nouvelles solutions

Pour pouvoir faire évoluer le processus, il est nécessaire de générer de nouvelles solutions à partir de la population référence. Elles sont obtenues à partir de combinaisons de sous-ensembles des solutions de la population et sont alors candidates pour l'intégrer à leur tour. A chaque itération de l'algorithme une partie au moins des solutions références doit être modifiée. Dans le cas contraire la population a convergé. Nous utilisons quatre types de sous-ensembles obtenus à l'aide du composant (C4) :

- Les sous-ensembles de type 1 : toutes les solutions de la population référence sont considérées deux à deux.
- Les sous-ensembles de type 2 : chaque sous-ensemble de type 1 est augmenté en ajoutant la meilleure solution non encore présente.
- Les sous-ensembles de type 3 : chaque sous-ensemble de type 2 est augmenté en ajoutant la meilleure solution non encore présente.
- Les sous-ensembles de type 4 : il s'agit de prendre les *i* meilleures solutions, pour *i* allant de 5 à la taille de la population référence.

4.4.1 Détermination des sous-ensembles

Les sous-ensembles générés sont de taille 2, 3 et 4 pour les types 1, 2 et 3 respectivement. Pour le dernier type nous avons au moins 5 solutions et au plus le nombre de solutions de la population référence. La définition de ces quatre types de sous-ensembles peut être justifiée comme suit. Le type 1 représente la combinaison la plus simple possible. Ce genre de combinaison apparaît dans les algorithmes génétiques (combinaison entre deux parents). Les sous-ensembles de type 2 et 3 sont des extensions du type 1. Ils permettent d'apporter plus d'informations lors de la combinaison en augmentant le nombre de parents. Il faut cependant pouvoir définir un processus de combinaison adapté qui prenne en compte ces caractéristiques. Le fait que dans les sous-ensembles de type 2 à 4 la meilleure solution au moins doit être présente permet aussi de limiter le nombre total de sous-ensembles.

Au cours de l'algorithme, deux stratégies peuvent être utilisées pour déterminer les sous-ensembles : une gestion statique ou une gestion dynamique de la population (Glover [Glo98]). Une gestion statique des sous-ensembles consiste à générer à chaque itération tous

les sous-ensembles associés à la population. L'algorithme 4.5 décrit cette approche pour les sous-ensembles de type 1. Dans cet algorithme, et dans la suite de ce chapitre, *Ref* est la population référence. Nous notons sa taille par |*Ref*|, et *Ref*[*i*] dénote la solution placée à la $i^{\text{ième}}$ position dans *Ref*. *X* est un sous-ensemble de solutions, et *S*(*X*) représente l'ensemble des solutions (au moins une) généré par la méthode de combinaisons appliquée au sous-ensemble *X*.

Algorithme 4.5 Génération des sous-ensembles de type 1 dans une version statique

Entrée : la population *Ref*.
Sortie : la population *Ref*.

```
1 : pour i allant de 1 à |Ref| - 1 faire
2 :   pour j = i + 1 à |Ref| faire
3 :     Poser X = {Ref[i],Ref[j]};
4 :     Appliquer la méthode de combinaison sur X et récupérer S(X)
5 :     Appliquer la méthode d'amélioration à S(X)
6 :     Appeler la méthode de mise à jour de Ref sur S(X)
7 :   fin pour
8 : fin pour
```

Avec cette approche le nombre de sous-ensembles est relativement important, mais il faut surtout remarquer qu'il est inutile de tous les générer à chaque itération. En effet lorsqu'un sous-ensemble est obtenu au cours de la recherche, il n'y a pas d'intérêt à le créer de nouveau par la suite puisque la solution obtenue par la combinaison sera toujours la même (excepté si nous utilisons plusieurs combinaisons qui changent au cours du processus). C'est pourquoi une gestion dynamique de la population référence est souhaitable. Ce point est un autre élément qui différencie la recherche dispersée des algorithmes génétiques dans lesquels il n'y a généralement pas de moyens mis en œuvre pour contrôler la répétition des solutions générées. La population référence est elle-même de nature dynamique puisque des solutions changent à chaque itération. Il apparaît donc normal de mettre en oeuvre une méthode qui génère les sous-ensembles en prenant en compte cette caractéristique. Cela permet aussi de générer moins de sous-ensembles au cours de la recherche.

Dans toute la suite de ce chapitre nous utilisons une gestion dynamique de la population pour la génération des sous-ensembles. Cette stratégie dynamique repose sur deux vecteurs permettant de mémoriser à quelle itération a été modifiée chaque solution dans la population pour la dernière fois (*changement*), et à quelle itération a été appliquée la méthode de génération pour chaque type de sous-ensembles (*lancement*). Il est facile de savoir si la solution mémorisée à la place *i* de la population a été modifiée depuis le dernier appel à la génération des sous-ensemble de type *j* en vérifiant si *changement*[*i*] ≥ *lancement*[*j*]. Si cette

condition est vérifiée, alors la solution placée à la $i^{\text{ième}}$ place dans la population doit être considérée comme nouvelle pour ce type de sous-ensembles. Nous supposons dans la suite que la population est organisée de manière à ce que les solutions soient triées selon l'ordre décroissant de leur coût

$$v(Ref[1]) \geq v(Ref[2]) \geq \ldots \geq v(Ref[|Ref|]) \quad (4.1)$$

Nous présentons dans l'algorithme 4.6 le fonctionnement du « gestionnaire » pour la génération des sous-ensembles. Il consiste à déterminer quelles sont les nouvelles solutions avant d'appeler le programme de génération des sous-ensembles correspondants à l'itération courante. Ce processus est répété tant que de nouvelles solutions sont détectées dans la population. Celle-ci est en entrée de l'algorithme, comme l'itération courante et les deux tableaux cités précédemment. La ligne 4 de l'algorithme 4.6 permet de déterminer l'ensemble des indices des nouvelles solutions depuis le dernier lancement de la génération des sous-ensembles du type courant. La ligne 8 correspond à un appel de l'algorithme de génération des sous-ensembles. Les mises à jour éventuelles de la population, du tableau *changement* et de *iter* se font au sein de cet algorithme. Ajoutons que d'un point de vue algorithmique le paramètre *iter* vaut 0, et les deux tableaux *changement* et *lancement* sont remplis de 0 lors du premier appel de l'algorithme. Toutes les solutions de la population sont ainsi initialement considérées comme nouvelles.

Algorithme 4.6 Algorithme de contrôle pour la génération des sous-ensembles

Entrées : la population *Ref* ; l'itération courante *iter*; *changement*; *lancement*.
Sorties : la population, *Ref* ; l'itération courante *iter*; *changement*; *lancement*.

```
 1 : stop = faux ; type = 1;
 2 : tant que stop = faux faire
 3 :    iter = iter + 1;
 4 :    I_nouvelles = {i ∈ {1,…,|Ref| }, changement[i] ≥ lancement[type] };
 5 :    I_anciennes = { i ∈ {1,…,|Ref|, i ∉ I_nouvelles};
 6 :    si |I_nouvelles| = 0 alors stop = vrai;
 7 :    sinon
 8 :       générer_type(Ref, I_nouvelles, I_anciennes, type, iter);
 9 :       lancement[type] = iter ;
10 :       type = type + 1 ;
11 :       si type > 4 alors type = 1 ;
12 :    fin si
13 : fin tant que
```

L'algorithme 4.7 présente une méthode pour générer les sous-ensembles de type 1 de manière dynamique. Les autres procédures de génération des sous-ensembles suivent le même

principe. Il convient cependant de les modifier pour s'assurer que la (ou les) meilleures solutions soient présentes dans chaque sous-ensemble.

Algorithme 4.7 Génération des sous-ensembles de type 1 en version dynamique

Entrées : la population *Ref* ; les indices des nouvelles *I_nouvelles*; les indices des anciennes *I_anciennes*; l'itération courante *iter*.
Sortie : la population *Ref*.

```
 1 : pour j ∈ I_nouvelles faire
 2 :    pour k ∈ I_nouvelles faire
 3 :       Poser X = {Ref[j],Ref[k]};
 4 :       S(X) = combiner(X);
 5:        Appliquer la méthode d'amélioration à S(X);
 6 :       Appliquer la méthode de mise à jour de Ref avec S(X);
 7 :    fin pour
 8 :    pour k ∈ I_anciennes faire
 9 :       Poser X = {Ref[j],Ref[k]};
10 :       S(X) = combiner(X);
11 :       Appliquer la méthode d'amélioration à S(X);
12 :       Appliquer la méthode de mise à jour de Ref avec S(X);
13 :    fin pour
14 : fin pour
```

Une dernière question reste à aborder pour la mise à jour de la population référence au cours de la génération des nouvelles solutions. Dans l'algorithme 4.7 précédent nous supposons que la population est maintenue à jour après chaque combinaison. Cela implique une gestion dynamique de la population pendant les combinaisons. Ce point est différent de la gestion dynamique dont nous avons déjà parlé. Il s'agit ici de considérer qu'une nouvelle solution est candidate pour intégrer la population immédiatement après sa génération. Ce type de méthode suggéré par Glover [Glo98] peut être vu comme étant plus « agressif » qu'une méthode statique qui consiste à stocker toutes les solutions obtenues lors des combinaisons avant de faire une mise à jour globale de la population. Nous avons testé les deux approches et nous donnons nos conclusions dans la section 4.5 sur les expériences numériques.

4.4.2 Combinaisons des sous-ensembles

Le composant (C5) consiste à définir la ou les procédures mises en œuvre pour créer les nouvelles solutions à partir d'un sous-ensemble généré par (C4). Nous utilisons plusieurs combinaisons dans notre algorithme. La première est une combinaison linéaire entre les solutions du sous-ensemble. Elle repose sur la notion de *score* utilisée par exemple par

Laguna et Marti [LM03]. Le principe est de calculer un score s_j pour chaque variable x_j de manière à déterminer si elle doit être fixée à 1 (si son score est > à 0.5) ou non comme suit

$$s_j = \frac{\sum_{x \in S} cx \times x_j}{\sum_{x \in S} cx} \tag{4.2}$$

avec x qui représente une solution dans le sous-ensemble S.

Nous illustrons le fonctionnement de cette méthode dans l'exemple suivant.

Exemple 4.3 : illustration de la combinaison score.

Nous considérons le problème $P2$ de l'exemple 4.2. Supposons que le sous-ensemble S soit composé des quatre solutions suivantes :

$x^1 = (1\ 0\ 0\ 0\ 0)$ $cx^1 = 5$; $x^2 = (0\ 1\ 0\ 1\ 0)$ $cx^2 = 5$;

$x^3 = (0\ 1\ 1\ 0\ 0)$ $cx^3 = 6$; $x^4 = (1\ 0\ 1\ 0\ 1)$ $cx^4 = 8$;

Les scores obtenus pour les 5 variables du problème sont donnés pour chaque solution dans le tableau suivant, et on a $\sum_{x \in S} cx = 24$. Chaque ligne dans ce tableau correspond à une solution et chaque colonne à une variable.

	x_1	x_2	x_3	x_4	x_5
x^1	5/24	0	0	0	0
x^2	0	5/24	0	5/24	0
x^3	0	6/24	6/24	0	0
x^4	8/24	0	8/24	0	8/24
Total	0.54	0.46	0.58	0.21	0.33

La solution obtenue par la combinaison est donc (1 0 1 0 0).

Nous référençons dans la suite cette combinaison par l'appellation *score*. Il est clair qu'avec ce genre de combinaison il est possible de générer une solution appartenant au sous-ensemble initial. Cela est d'autant plus le cas lorsque la taille du sous-ensemble est petite.

Nous utilisons également une méthode de croisement comme dans les algorithmes génétiques pour les sous-ensembles de type 1. Le point de croisement est ici l'indice $n/2$. D'autres approches plus sophistiquées pour choisir ce point auraient pu être testées mais cette combinaison, que nous référençons par *croisement* dans la suite, a surtout été utilisée lors de l'étude expérimentale présentée dans la section 4.5.1.

Nous définissons aussi deux approches basées sur la méthode des chemins reliants. La première version de l'algorithme peut être vue comme une méthode « directe », alors que la seconde définit un chemin plus détourné entre les deux solutions. Le but de la méthode des

chemins reliants est de générer un chemin entre une solution origine et une solution destination. Elle existe sous des formes sophistiquées dans lesquelles des concepts d'intensification ou de diversification sont utilisés. Nos deux algorithmes sont des versions simples de chemins. Le but est d'être assez rapide car nous les utilisons pour combiner les sous-ensembles de type 1.

Nous décrivons dans l'algorithme 4.8 le fonctionnement de la première méthode. Nous l'appelons *chemin1* dans la suite de ce chapitre. Dans cet algorithme et le suivant nous notons par e_j le vecteur de dimension n dans lequel seul l'indice j est fixé à 1. La solution x est la solution origine et la solution y est la solution destination. La première boucle *pour* (ligne 3) consiste à retirer chacune des variables égale à 1 dans la solution origine et égale à 0 dans la solution destination et à la remplacer par une variable initialement à 0 dans la solution origine en maximisant la valeur de l'objectif (il s'agit d'un principe glouton). Dans la seconde boucle *pour* (ligne 10) nous essayons d'ajouter les variables restantes dans la solution. Si plus aucune variable ne peut être ajoutée (ligne 15) le processus est arrêté. Cela peut se produire si la solution destination n'est pas réalisable par exemple.

Algorithme 4.8 Méthode de chemin direct entre deux solutions

Entrées : les deux solutions x et y.
Sortie : la meilleure solution x^* distincte de x et y.

1 : $F^0 = \{j : x_j \neq y_j, x_j = 0\}$; $F^1 = \{j : x_j \neq y_j, x_j = 1\}$;
2 : $z = x$; $cx^* = -\infty$;
3 : **pour** $j \in F^1$ **faire**
4 : $k^* = \arg\max_{k \in F^0} \{c_k : A(z + e_k - e_j) \leq b\}$;
5 : **si** k^* existe **alors**
6 : $z = z + e_{k^*} - e_j$; $F^1 = F^1 - \{j\}$; $F^0 = F^0 - \{k^*\}$;
7 : **si** $cz > cx^*$ **alors** $x^* = z$;
8 : **sinon** $z = z - e_j$; $F^1 = F^1 - \{j\}$;
9 : **fin pour**
10 : **pour** $j \in F^0$ faire
11 : $k^* = \arg\max_{k \in F^0} \{c_k : A(z + e_k) \leq b\}$;
12 : **si** k^* existe **alors**
13 : $z = z + e_{k^*}$; $F^0 = F^0 - \{j\}$;
14 : **si** $cz > cx^*$ et $z \neq y$ **alors** $x^* = z$;
15 : **sinon** stop (plus possible d'ajouter de variables)
16 : **fin pour**
17 : retourner x^*

Notons que nous pouvons aussi définir un ordre pour le parcours des variables plutôt que d'utiliser un principe glouton. Nous conservons dans l'algorithme 4.8 la meilleure solution générée (en ne considérant pas les solutions source et origine comme des solutions du chemin). Lorsque nous effectuons un chemin entre une solution réalisable et une solution non réalisable, nous stoppons le processus dès que nous arrivons dans le domaine non réalisable. De même si nous effectuons un chemin entre une solution non réalisable et une solution réalisable, nous commençons par projeter la solution de départ le plus près possible de la frontière.

L'algorithme 4.9 donne une description de la seconde approche, appelée *chemin2* dans la suite.

Algorithme 4.9 Méthode de chemin étendu

Entrées : les deux solutions x et y.
Sortie : l'ensemble des solutions générées sur le chemin *pool*.

1 : $F^0 = \{j : x_j \neq y_j, x_j = 0\}; F^1 = \{j : x_j \neq y_j, x_j = 1\};$
2 : $z = x$; $P = \varnothing$; $J = \varnothing$;
3 : **pour** $j \in F^1$ **faire**
4 : $k^* = \arg\max_{k \in F^0} \{c_k : A(z + e_k - e_j) \leq b\}$;
5 : **si** k^* existe **alors** $z = z + e_{k^*} - e_j$;
6 : **sinon** $z = z - e_j$; $J = J + \{j\}$;
7 : ajouter z à P; $F^1 = F^1 - \{j\}$; *fin* = faux;
8 : **tant que** non fin **faire**
9 : $k^* = \arg\max_{k \in N - J} \{c_k : z_k = 0; A(z + e_k) \leq b\}$;
10 : **si** k^* existe **alors** $z = z + e_{k^*}$; **si** $y_{k^*} = 0$ **alors** $F^1 = F^1 + \{k^*\}$;
11 : **sinon** *fin* = vrai;
12 : ajouter z à P;
13 : **fin tant que**
14 : **fin pour**
15 : retourner P

La différence majeure par rapport à l'algorithme précédent est la définition des candidats à un ajout dans la solution. Nous considérons en effet en plus des variables appartenant à l'ensemble F^0 celles qui sont égales à 0 dans les deux solutions x et y. L'utilisation de l'ensemble J dans cet algorithme permet d'assurer la convergence du processus en interdisant l'ajout d'une variable déjà retirée de la solution courante. Dans cet algorithme nous conservons toutes les solutions générées sur le chemin dans la variable P. Il est facile d'adapter l'algorithme pour ne conserver qu'une partie des solutions ou la meilleure.

Nous avons finalement implémenté une méthode appelée méthode des chemins en étoile (*star paths* en anglais) proposée par Glover [Glo95]. Cette approche a été utilisée pour construire un ensemble de solutions diverses par Glover, Lokketangen et Woodruff [GLW00]. Un chemin en étoile est défini comme étant une suite de solutions entières entre une solution source et une solution destination. Une solution appelée *référence* (pas nécessairement entière) est également utilisée pour définir la direction du chemin. Les variables des solutions intermédiaires sont fixées en fonction de ces trois solutions et d'une valeur réelle (paramètre de l'algorithme). Cette valeur permet d'obtenir plusieurs chemins en étoile distincts entre deux solutions données. Nous avons appliqué une partie de l'algorithme décrit dans [GLW00] avec une solution optimale de la relaxation en continu du problème comme référence et la valeur 0 comme paramètre. Notre algorithme construit un chemin en étoile entre deux solutions passées en entrée. Nous ne rentrons pas plus dans le détail de l'implémentation. Le lecteur peut se référer à un des deux articles précédents pour plus de précisions.

4.5 Expériences numériques

Nous avons développé notre algorithme de recherche dispersée à partir des principes présentés dans les sections précédentes, ainsi qu'au regard des résultats obtenus lors d'un ensemble de tests préliminaires. Ces expériences numériques ont pour but de nous guider sur les meilleures stratégies à adopter pour le problème *SADM* en particulier. Une autre motivation est de vérifier et compléter les *leçons* proposées par Laguna et Armentano [LA05]. Ces leçons sont tirées de constatations faites lors de la résolution de problèmes d'optimisation avec la recherche dispersée.

Tous les algorithmes présentés dans ce chapitre ont été codés en langage 'C' et compilés avec « gcc » et l'option –O2. La machine que nous avons utilisée est équipée d'un processeur Pentium IV 3,4Ghz avec 4Go de RAM. La première partie de cette section est consacrée à notre étude expérimentale. Nous abordons ensuite les résultats finaux de l'algorithme sur les instances difficiles existantes du problème *SADM*.

4.5.1 Influence des composants de l'algorithme

Afin d'évaluer l'influence des composants de l'algorithme il est nécessaire de définir une version initiale simple. Nous la définissons en respectant les points suivants.

a. Utilisation du générateur dichotomique pour obtenir une population initiale (en référence aux résultats de la section 4.2).

b. Amélioration des solutions par recherche locale (selon le principe décrit dans la section 4.3).
c. Gestion dynamique de la population référence pour la génération des sous-ensembles (section 4.3).
d. Combinaison de type croisement pour les sous-ensembles de type 1.
e. Combinaison basée sur le score pour les autres sous-ensembles.
f. Gestion statique de la mise à jour de la population référence pendant les combinaisons.
g. La taille de la population référence est fixée à 40 solutions.

Nous utilisons un ensemble d'instances de taille moyenne de problèmes *SADM* appartenant à la même classe de manière à pouvoir mesurer l'impact de chaque composant. Nous choisissons l'ensemble des 30 instances tirées des instances classiques mentionnées dans la section 4.2 appelées *weish* dans la littérature. Ces instances comportent de 30 à 90 variables pour 5 contraintes. Cinq instances distinctes ont été générées pour chaque valeur de n = 30, 40, 50, …, 90. Il est clair que les instances choisies ne représentent pas tous les types d'instances de problèmes *SADM* possibles car elles ont été générées suivant une règle précise. Cette étude expérimentale a pour but de donner des indications sur les meilleurs choix stratégiques à faire. Ils doivent ensuite être confirmés lors des expériences numériques sur les plus grandes instances.

Les tests que nous avons réalisés visent à répondre aux quatre questions suivantes qui sont abordées dans les sections qui suivent.

1. Quel type de population de solutions est le plus intéressant à utiliser selon plusieurs critères : la diversité, la qualité des solutions et la convergence de la population en nombre d'itérations ?
2. Quelle est l'influence des combinaisons et de la mémoire sur le processus de recherche ?
3. Le fait d'intégrer des phases d'intensification peut-il apporter un plus à la recherche ?
4. Une gestion dynamique de la mise à jour de la population référence pendant les combinaisons est-elle bénéfique ?

4.5.1.1 Division de la population et critère d'ajout

La première série de tests évalue l'impact du type des solutions dans la population référence. Nous étudions aussi plusieurs critères d'ajout des solutions. Trois types de

solutions sont utilisées : des solutions élites, des solutions diverses et des solutions non réalisables. Au niveau des critères d'ajout, nous retenons les possibilités suivantes. Les solutions diverses sont toujours ajoutées selon leur diversité lors de la phase initiale. Elles peuvent ensuite être ajoutées selon leur coût ou leur diversité lors de l'algorithme de recherche dispersée. La gestion de la population des solutions élites est simple et intuitive. Elle consiste à vérifier si la nouvelle solution rencontrée est meilleure qu'une autre solution de la population au moins et qu'elle n'est pas déjà présente. Finalement, la gestion des solutions non réalisables est similaire à celle des solutions diverses. Nous conservons pour chaque solution x une valeur notée $violation(x)$ qui représente la somme des violations de la solution x sur l'ensemble des contraintes du problème, de manière à estimer l'éloignement de la solution par rapport au domaine réalisable. La valeur associée à une solution x est déterminée comme suit

$$violation(x) = \sum_{i \in M} \min\{0 ; b_i - A_i x\}.$$

Nous conservons uniquement les solutions les plus proches de la frontière entre les domaines réalisable et non réalisable.

Nous testons l'algorithme de recherche dispersée sans limite d'itérations. Le processus s'arrête donc dès que la population converge. Nous avons 6 populations références différentes définies comme suit :

- *Meilleure* : une population composée uniquement de solutions élites.
- *Div_D* : une population composée uniquement de solutions diverses (sauf la meilleure qui est conservée). L'ajout dans la population se fait selon la diversité à la fois durant la phase initiale et durant l'algorithme de recherche dispersée.
- *Div_C* : même principe mais l'ajout dans la population se fait selon la qualité des solutions pendant l'algorithme de recherche dispersée.
- *MD_D* : une population composée d'une moitié de solutions élites et d'une moitié de solutions diverses. Les solutions diverses sont ajoutées selon la diversité dans l'algorithme de recherche dispersée.
- *MD_C* : même principe que *MD_D* mais l'ajout des solutions diverses repose sur le coût au cours de l'algorithme de recherche dispersée.
- *MDNr* : une population composée de solutions élites, diverses et non réalisables (2/5, 2/5, 1/5). L'ajout des solutions diverses se fait toujours selon la diversité. Pour pouvoir comparer la diversité de cette population avec les autres, nous utilisons ici 20 solutions

élites, 20 solutions diverses et 10 solutions non réalisables (non prises en compte pour la diversité).

Pour mesurer les performances de chaque population, nous utilisons les mesures Δd^i et Δc^i présentées dans la section 4.2. Nous représentons ces mesures dans les figures suivantes par *deltaD* et *deltaC* qui sont des moyennes sur les 30 instances. La figure 4.3 présente les résultats obtenus après l'application de la phase initiale. Comme l'algorithme de recherche dispersée n'est pas appliqué, les algorithmes pour lesquels seul le critère d'ajout des solutions diverses change au cours de la recherche dispersée obtiennent les mêmes résultats. Nous présentons donc uniquement les résultats obtenus pour les populations *Meilleure*, *Div_D* et *MD_D*. La courbe en pointillés avec des triangles donne la valeur $\Delta d^i + \Delta c^i$. Plus cette valeur est grande, et plus la population est performante selon les deux critères mesurés.

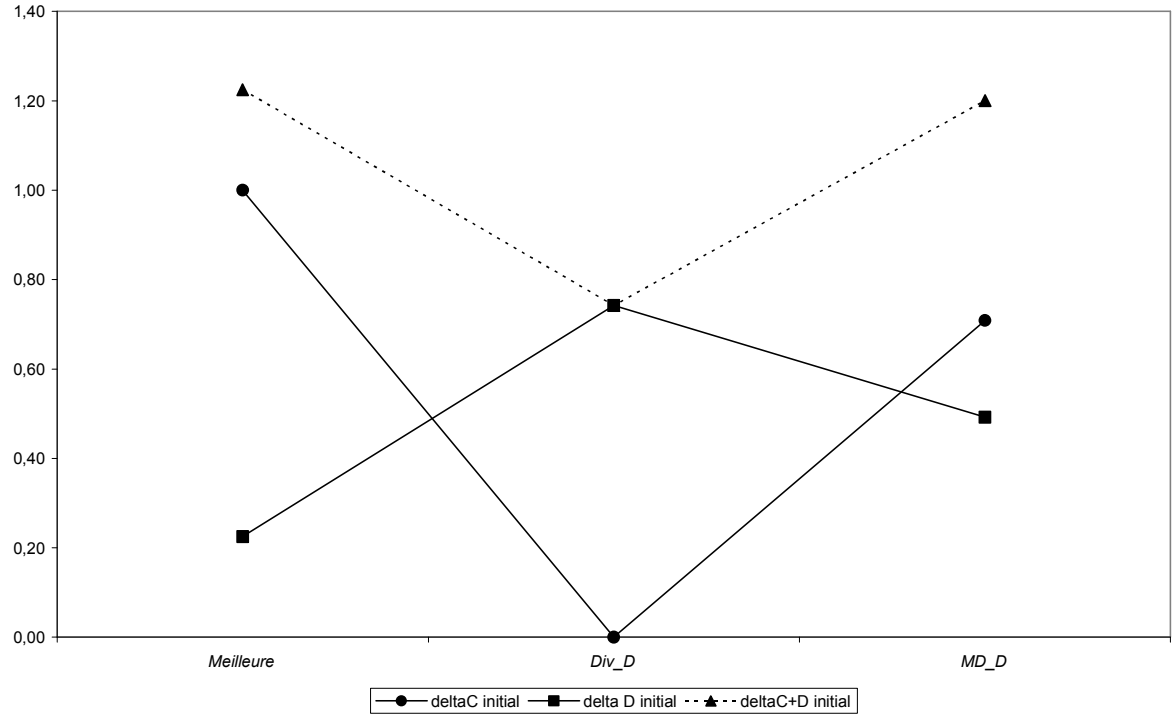

Figure 4.3 – Diversité et qualité après la phase initiale.

La figure 4.3 montre clairement que l'utilisation d'une population composée uniquement de solutions diverses est nettement dominée car la valeur Δc^i associée est nulle. Même si la valeur Δd^i est la plus élevée, le compromis entre les deux critères ne paraît pas intéressant. Les deux autres populations permettent d'obtenir des résultats du même ordre au niveau de la valeur $\Delta d^i + \Delta c^i$, même si la population utilisant des solutions élites et diverses est plus « régulière » sur les deux critères.

La figure 4.4 présente les résultats obtenus lorsque nous appliquons l'algorithme de recherche dispersée simple.

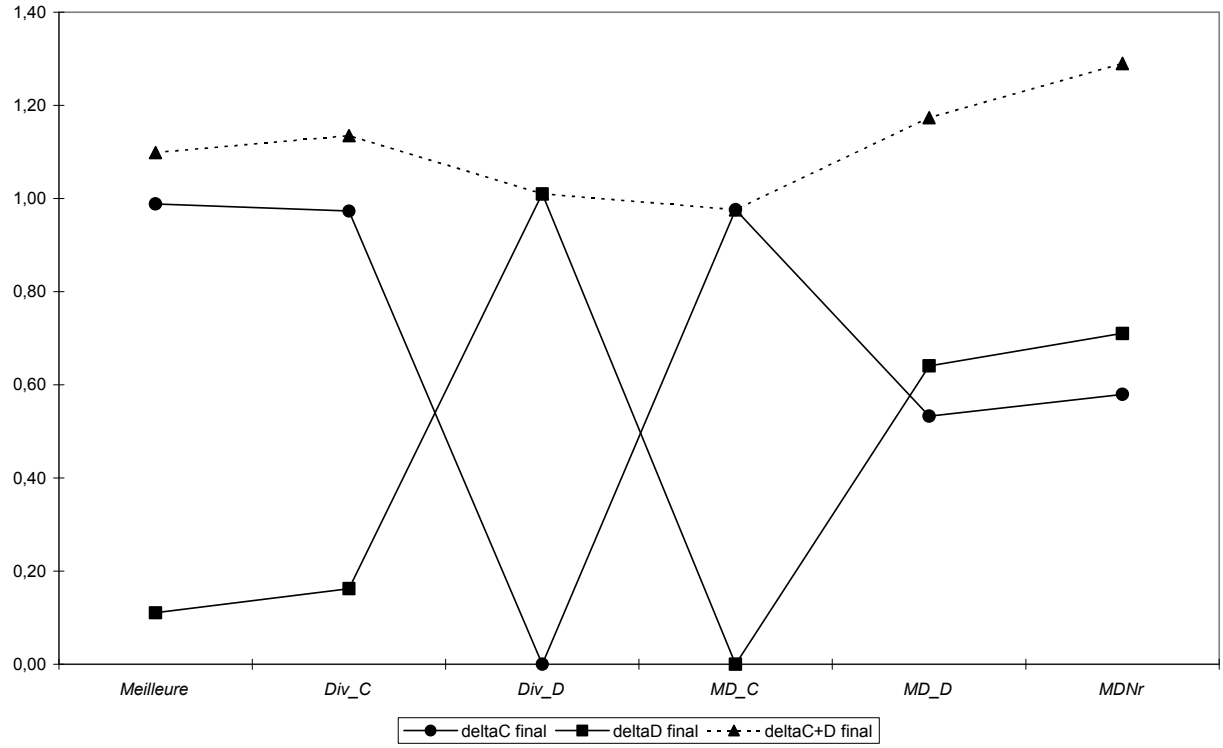

Figure 4.4 – Diversité et qualité après la recherche dispersée.

Plusieurs enseignements peuvent être tirés de la figure 4.4. Nous pouvons tout d'abord voir que la conclusion précédente sur la population composée uniquement de solutions diverses est en partie confirmée ici. La population *Div_D* est en effet une des deux populations ayant la valeur $\Delta d^i + \Delta c^i$ la plus faible. Il semble plus intéressant d'utiliser un ajout selon le coût lors de la recherche dispersée de manière à améliorer fortement la qualité de la meilleure solution dans le cas d'une population diverse. Si nous utilisons des solutions élites et diverses il paraît au contraire plus intéressant d'ajouter les solutions diverses selon la diversité pendant la recherche dispersée. Une population formée uniquement de solutions élites entraîne logiquement une qualité importante des solutions mais une diversité faible. Finalement, la population basée sur les solutions élites, diverses et non réalisables permet d'obtenir deux valeurs de Δd^i et Δc^i proches de 0.6 pour une valeur de $\Delta d^i + \Delta c^i$ supérieure à 1. Cette approche est la plus performante ici, même si les résultats sont proches de ceux de la population *MD_D*.

Nous donnons dans la figure 4.5 le saut moyen (en %) par rapport à la valeur optimale du problème de la meilleure solution (après la phase initiale et après la phase de recherche dispersée). Nous précisons aussi le nombre moyen d'itérations effectuées. L'axe de gauche du graphique correspond au saut initial et au nombre d'itérations moyen effectuées. L'axe de droite correspond au saut final. Le saut de la meilleure solution après la phase initiale est logiquement important (entre 40% et 50% de l'optimum), le but étant d'obtenir une population diverse. L'application de la recherche dispersée permet une nette amélioration de la qualité et le saut final est compris entre 1.9% (pour *MDNr*) et 7.7% (pour la population *Div_D*).

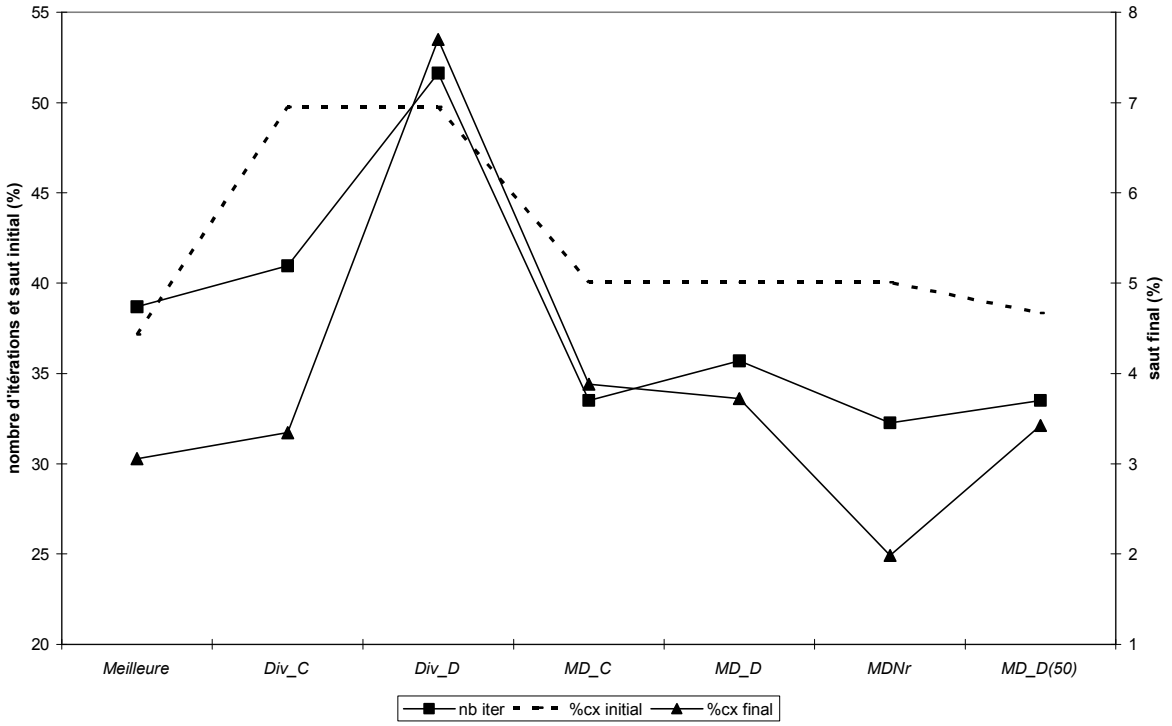

Figure 4.5 – Qualité de la solution et convergence.

Nous pouvons également voir dans la figure 4.5 que la population *Div_D* permet de construire des phases de recherche plus longues (ici de l'ordre de 52 itérations), mais la qualité de la solution finale est moins bonne. La moyenne du nombre d'itérations est comprise entre 32 et 40 itérations pour les populations les plus performantes. La convergence de l'algorithme s'accélère donc logiquement lorsque l'accent est mis sur les solutions élites. Notons que nous précisons également dans ce graphique les résultats obtenus par une population composée de 25 solutions élites et 25 solutions diverses (*MD_D*(50)). Cela nous permet de justifier l'impact positif des solutions non réalisables sur le processus de recherche. En effet les résultats de la population *MD_D*(50) montrent que le saut moyen de la solution

finale n'est pas amélioré dans la population *MDNr* uniquement grâce à un nombre plus important de solutions (et donc de combinaisons) (1.9% au lieu de 3.4%).

Ces résultats nous conduisent à l'élimination de la population composée uniquement de solutions diverses avec un ajout basé sur la diversité, et de celle divisée en deux (solutions élites et diverses) avec l'ajout selon le coût. Il reste quatre versions donnant les meilleurs résultats. Nous testons dans la suite notre algorithme dans le cas où le processus de recherche est relancé une fois que la population converge. Nous appliquons une leçon de Laguna et Armentano [LA05] qui consiste à utiliser un générateur de solutions diverses pour obtenir une nouvelle population (nous conservons uniquement la meilleure solution). Le générateur est appliqué à partir d'une des solutions présentes dans la population avant la reconstruction jusqu'à avoir généré $5 \times n$ solutions.

La figure 4.6 présente les résultats obtenus pour le saut moyen de la solution finale et le nombre d'itérations moyen entre deux reconstructions de la population (la diversité n'est plus mentionnée car elle dépend de la convergence des algorithmes et ainsi du nombre d'applications de l'algorithme de recherche dispersée).

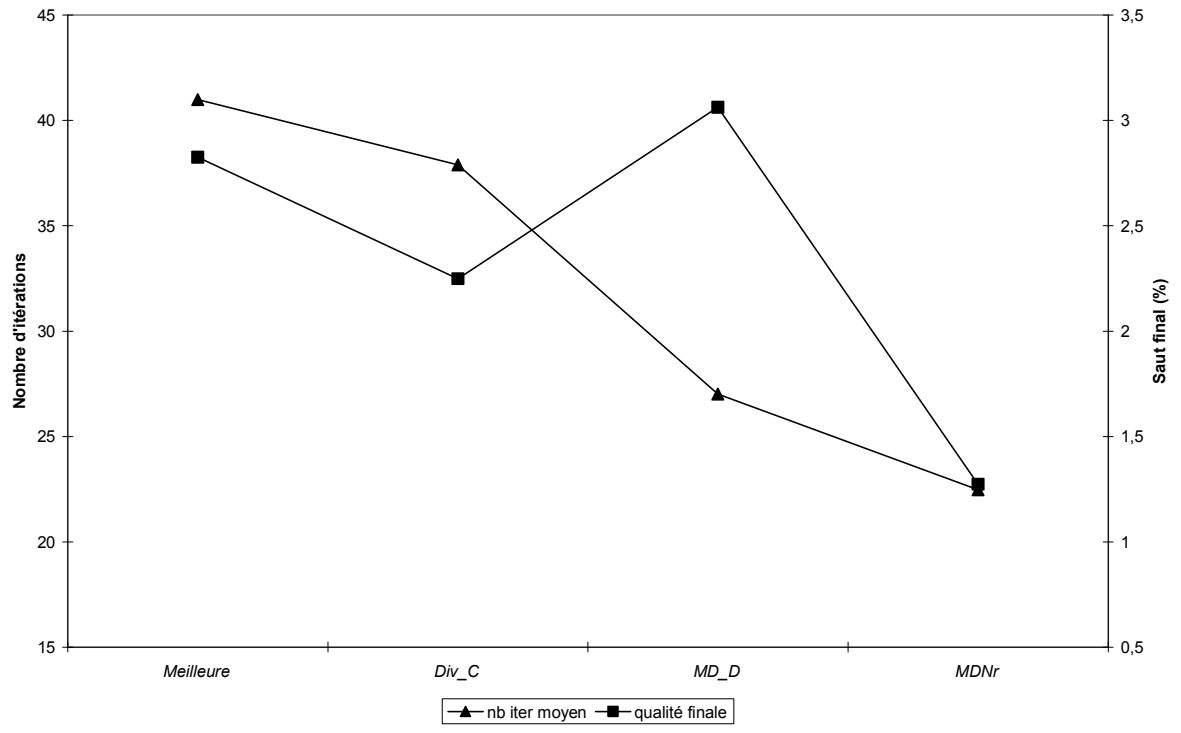

Figure 4.6 – Qualité des solutions et convergence en répétant le processus.

Le nombre d'itérations moyen d'une phase de recherche dispersée varie fortement (entre 22 et 41). Nous pouvons noter que la population *MDNr* est celle qui converge le plus vite. C'est également celle qui obtient la meilleure solution finale avec un saut moyen de

1.3%. Les résultats obtenus par la population *Meilleure* confirment que l'utilisation de solutions diverses est nécessaire. Finalement la version la moins performante en terme de qualité finale est *MD_D*. Devant ce résultat décevant, nous avons testé la population *MD_C* précédemment éliminée. Cette version n'obtient pas de meilleurs résultats car le saut moyen final est de 3.4% avec un nombre d'itérations moyen de 29. Nous pouvons conclure que l'intérêt d'utiliser les solutions non réalisables est renforcé. La population *Meilleure* est à son tour éliminée. Les résultats obtenus par la population *Div_C* par rapport à ceux de *MD_D* nous font penser qu'il est intéressant d'utiliser un ensemble de solutions diverses plus grand que celui des solutions élites.

4.5.1.2 Impact des combinaisons et de la mémoire

Les résultats obtenus précédemment sont de qualité moyenne, surtout devant la taille des instances. Nous commençons cette partie en présentant les résultats obtenus lorsque nous remplaçons la combinaison de type croisement par la combinaison de type chemin1. Nous testons à nouveau plusieurs versions de l'algorithme en changeant la taille des sous-populations et/ou leur type selon les conclusions précédentes :

- *MD*(20-20), *MD*(15-25) et *MD*(10-30) : population de solutions élites et diverses (toujours ajoutées selon la diversité). Les nombres entre parenthèses indiquent le nombre de solutions élites et diverses respectivement.
- *MDNr*(15-15-10) et *MDNr*(15-20-5) : population avec solutions non réalisables. Les valeurs correspondent au nombre de solutions élites, diverses et non réalisables respectivement.
- *MDNr*(15-20-5)_CH : idem mais correspond au cas extrême dans lequel nous n'effectuons que des combinaisons de type 1. Nous prenons alors les solutions deux à deux uniquement et nous appliquons la combinaison chemin1.

Les résultats sont présentés dans la figure 4.7. Nous précisons le nombre moyen d'itérations entre deux reconstructions avec le premier rectangle et le temps d'exécution moyen en secondes pour chaque population avec le second. L'échelle correspondante est l'axe des ordonnées de gauche de la figure. La courbe précise le saut moyen de la solution finale et correspond à l'axe des ordonnées de droite.

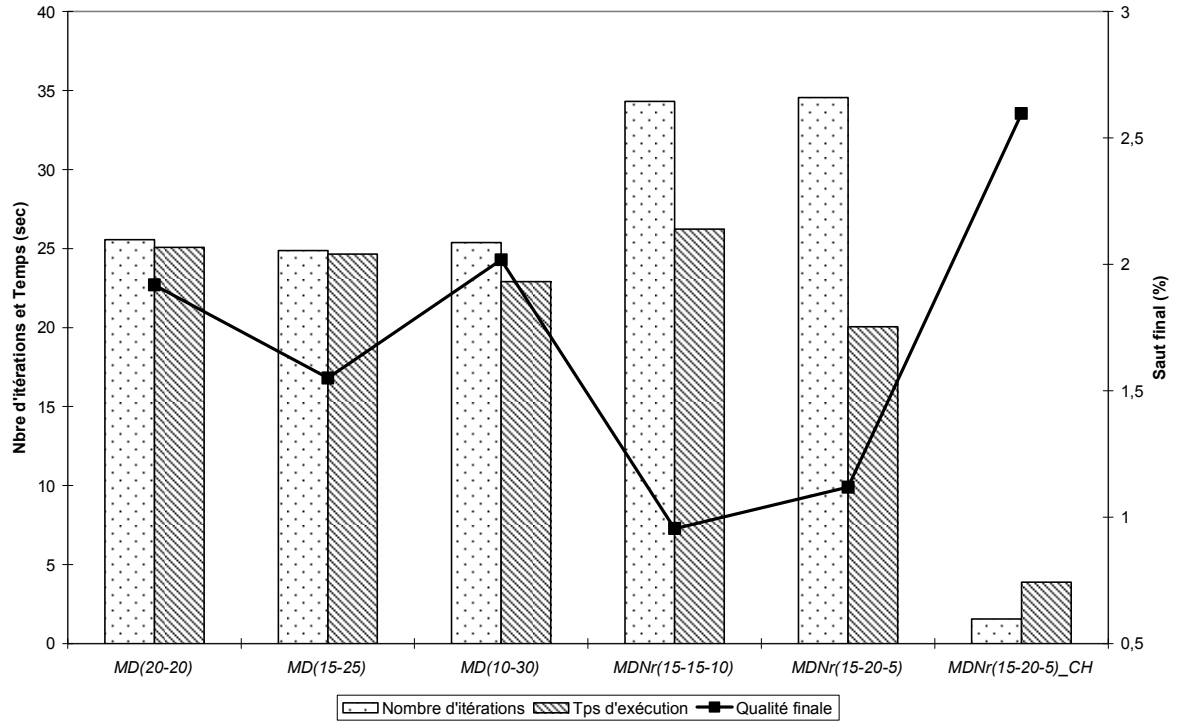

Figure 4.7 – Résultats obtenus selon la combinaison de type 1 utilisée.

Nous pouvons remarquer que l'utilisation d'un seul type de sous-ensembles n'est pas intéressante. L'algorithme est plus rapide (de l'ordre de 4 secondes en moyenne), mais le nombre d'itérations avant reconstruction de la population est très faible, et surtout la qualité de la solution finale est la moins bonne. Cela conforte à la fois le choix d'utiliser des sous-ensembles de différentes tailles et la leçon 13 de Laguna et Armentano selon laquelle l'utilisation de plusieurs méthodes de combinaison est généralement plus efficace. Les trois premières populations donnent des résultats proches. Le fait d'avoir plus de solutions diverses améliore un peu la qualité finale. Cela va dans le sens de la conclusion de la section 4.5.1.1 précédente. Il semble cependant qu'il ne faille pas avoir un déséquilibre trop important qui a tendance à inverser les résultats. Finalement les populations intégrant des solutions non réalisables obtiennent à nouveau les meilleurs résultats. La meilleure population obtient une solution finale en moyenne à 0.9% de l'optimum. Notons que le fait d'utiliser la combinaison de type chemin1 permet d'améliorer la qualité finale des solutions pour toutes les populations.

Pour améliorer le processus et guider la recherche, nous proposons d'intégrer l'utilisation de la mémoire dans l'algorithme. Nous conservons pour cela la somme des solutions ajoutées dans la population. Nous utilisons en particulier la fréquence des solutions élites (ou diverses lorsque nous n'avons qu'une solution élite) lors de la reconstruction de la

population. L'idée est de modifier l'ordre des variables utilisé dans les phases d'amélioration. En notant $freq_j$ la fréquence de chaque variable x_j, nous trions alors celles-ci selon la valeur $\Delta freq_j$ suivante

$$\Delta freq_j = \frac{freq_j - freq_{\min}}{freq_{\max} - freq_{\min}} \tag{4.3}$$

avec $freq_{\min}$ (resp. $freq_{\max}$) qui représente la plus petite (resp. grande) valeur des fréquences.

Ce nouveau tri des variables nous permet de visiter de nouvelles solutions et d'éviter de toujours obtenir les mêmes optima locaux lors des phases de recherche locale. Il tient compte de l'historique de la recherche et tend à favoriser les variables les plus souvent fixées à '1' lors des phases d'intensification. Nous utilisons l'ordre inverse lorsque nous diversifions la recherche de manière à privilégier les variables le moins souvent fixées à '1'. L'intégration de cette mémoire liée à la recherche tabou permet d'améliorer la qualité des solutions finales obtenues. En reprenant les notations de la figure 4.7 précédente, la solution finale de la population *MD*(15-25) est en moyenne à 0.67% de l'optimum, et celle de *MDNr*(15-20-5) à 0.64%. Nous pouvons noter la forte progression pour la population *MD*(15-25).

Nous pouvons également remplacer la combinaison de type chemin1 par celle du chemin en étoile. Celle-ci ne permet pas d'améliorer les résultats précédents. L'approche obtenue est plus rapide mais la qualité de la solution finale est généralement moins bonne. Finalement, l'utilisation de la combinaison de type chemin2 à la place de chemin1 permet d'obtenir l'ensemble des 30 solutions optimales avec les deux populations *MD*(15-25) et *MDNr*(15-20-5) précédentes. Le temps d'exécution de la méthode augmente (36 et 37 secondes en moyenne au lieu de 25 et 20). Cela s'explique par le fait que l'algorithme correspondant est plus coûteux puisque les chemins générés peuvent être plus longs.

Devant cette augmentation sensible du temps d'exécution, nous proposons la mise en place d'un mécanisme d'intensification pour les algorithmes utilisant les combinaisons chemin1 ou chemin2 pour les sous-ensembles de type 1.

4.5.1.3 Apport de l'intensification

L'intensification est un processus classique dans les algorithmes de recherche tabou comme nous l'avons vu dans les chapitres précédents. Nous appliquons un processus de résolution exacte sur un sous-problème de petite taille pour essayer de générer une solution de bonne qualité. Les variables sont triées en fonction de leur coût et de leur fréquence dans la

population (le tri est similaire à (4.3) en ajoutant le coût). Cela nous permet d'avoir un compromis entre l'historique de la recherche et l'objectif du problème. Nous essayons ensuite de fixer un certain nombre de variables à '1' (en fonction d'un paramètre) en respectant la réalisabilité de la solution. Le nombre de variables fixées à '1' est compris entre une borne inférieure et une borne supérieure sur la somme des variables d'une solution optimale du problème (voir chapitre 5, section 5.2.2 pour une description du principe). Une partie des variables (celles se situant au milieu selon le tri) est ensuite libérée en fonction d'un paramètre (nous utilisons la valeur 10 dans nos expériences numériques). Nous appliquons une résolution exacte sur le problème réduit obtenu. Si la solution générée peut être ajoutée dans la population alors nous relançons la recherche sans réinitialiser celle-ci.

Les résultats obtenus lorsque nous utilisons la combinaison de type chemin1 sont intéressants. Le saut moyen de la population *MD*(15-25) précédente est de 0.42% (au lieu de 0.67%). Pour la population *MDNr*(15-20-5), nous obtenons un saut moyen de 0.33% (au lieu de 0.64%). Nous parvenons à résoudre 19 instances sur les 30 de l'ensemble. Une autre manière d'intégrer des phases d'intensification est d'utiliser les algorithmes de chemins à la place de la résolution exacte. C'est en effet l'utilisation de ces algorithmes qui est la plus coûteuse dans le processus de recherche dispersée. Nous proposons dans ce cas d'utiliser le chemin en étoile comme combinaison pour les sous-ensembles de type 1. Une phase d'intensification correspond à appliquer un des deux algorithmes de chemins entre chaque couple de solutions de la population avant sa reconstruction. Nous testons les populations suivantes :

- *MD_I* et *MDNr_I* : les deux populations *MD*(15-25) et *MDNr*(15-20-5) décrites précédemment avec l'intensification basée sur la résolution exacte d'un problème réduit. La méthode chemin1 est utilisée comme combinaison pour les sous-ensembles de type1.
- *MD_SP* et *MDNr_SP* : le chemin en étoile remplace l'algorithme chemin1 qui devient le processus d'intensification.
- *MD_SP*2 et *MDNr_SP*2 : même principe mais avec l'algorithme chemin2 comme intensification.

Les résultats obtenus sont présentés dans la figure 4.8. Nous pouvons voir que la convergence moyenne est du même ordre pour toutes les populations. Nous pouvons aussi remarquer que l'intensification avec la méthode chemin1 ou la méthode chemin2 n'entraîne pas une augmentation du temps total moyen d'exécution. Les versions correspondantes

permettent d'obtenir les meilleurs résultats à la fois en terme de temps d'exécution et de qualité. La méthode chemin2 conserve un avantage par rapport à la méthode chemin1 puisque toutes les instances sont résolues. Le saut moyen final en utilisant le chemin1 est tout de même de 0.2%.

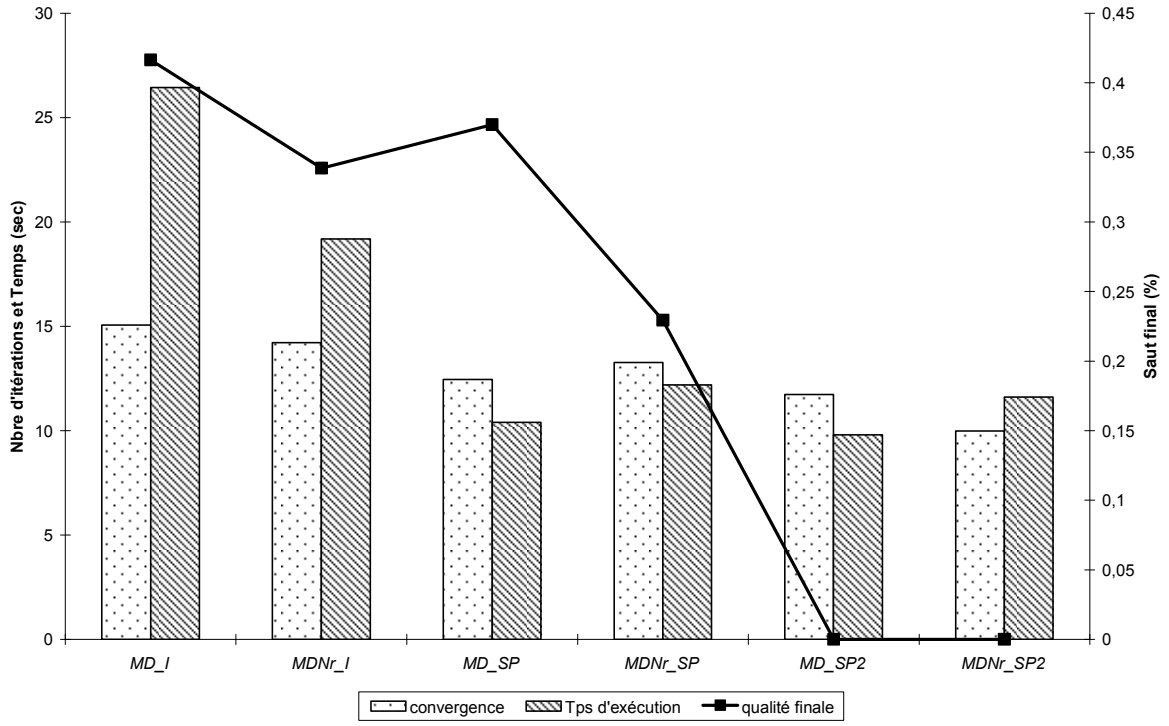

Figure 4.8 – Apport de l'intensification dans le processus.

L'ensemble de ces résultats montre clairement que l'intégration de la mémoire et de phases d'intensification au sein d'un algorithme de recherche dispersée enrichit celui-ci pour la résolution du problème du sac-à-dos multidimensionnel. Nous terminons cette étude expérimentale par la gestion de la mise à jour de la population référence pendant les combinaisons.

4.5.1.4 Gestion dynamique pendant les combinaisons

Comme nous l'avons vu à la fin de la section 4.4.1, deux stratégies sont possibles lors de la génération des nouvelles solutions par les combinaisons. La première approche, statique, est celle que nous avons utilisée jusqu'à présent. Nous présentons ici les résultats obtenus lorsque nous utilisons une gestion dynamique. Cette approche consiste à mettre à jour l'ensemble référence dès qu'une nouvelle solution est produite.

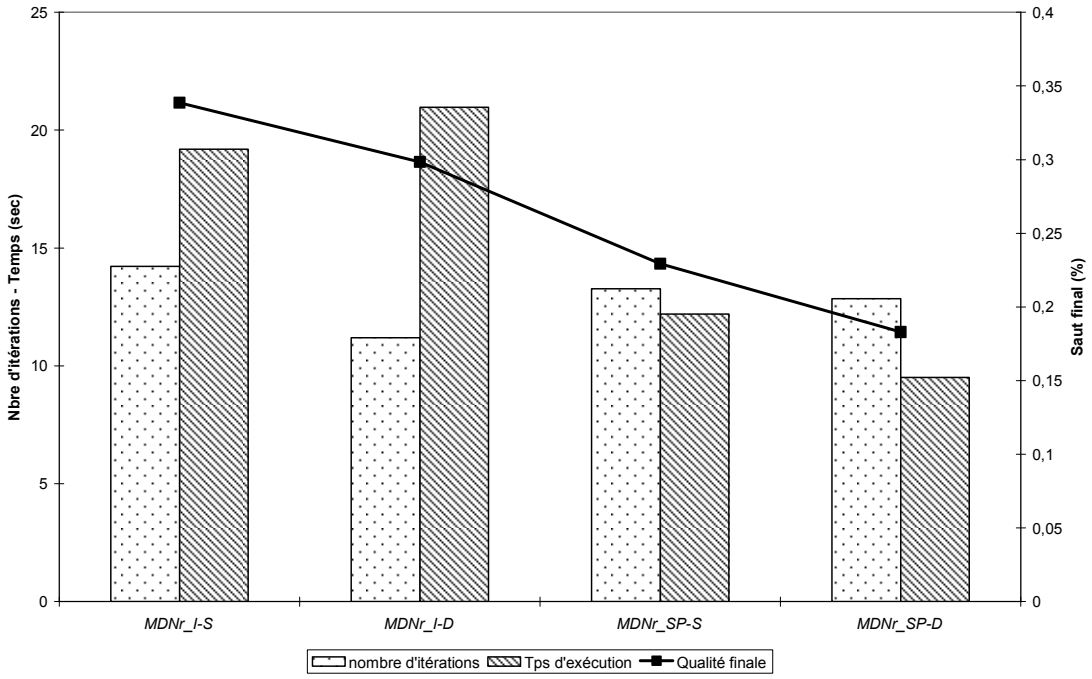

Figure 4.9 – Gestion Dynamique de la mise à jour des références.

Nous présentons les résultats obtenus dans la figure 4.9. Nous testons deux populations avec une stratégie statique (–S) et une stratégie dynamique (–D) : *MDNr_I* et *MDNr_SP*. La figure 4.9 montre qu'une gestion dynamique peut accélérer le processus de recherche. Ce n'est pas toujours le cas comme pour la version *MDNr_I*. L'autre point intéressant est que cette approche entraîne une amélioration de la qualité des solutions finales sur ces instances, même si elle est assez faible (diminution du saut de 0.33% à 0.3% pour *MDNr_I* et de 0.22% à 0.18% pour *MDNr_SP*). Les résultats numériques obtenus sur de plus grandes instances donneront plus d'éléments permettant de conclure sur l'impact de cette stratégie.

4.5.2 Résultats finaux et influence de la population initiale

Nous présentons dans la suite les résultats obtenus par quelques populations décrites précédemment sur les instances de la OR-Librairie [Bea90]. La recherche dispersée est une méthode évolutive. Pour évaluer notre algorithme nous proposons de comparer nos résultats avec deux algorithmes génétiques existants pour le problème *SADM*. Il s'agit de l'algorithme de Chu et Beasley [CB98] (colonnes *CB*) et de l'algorithme de Haul et Voss [HV98] (colonnes *HV*). Nous faisons aussi une comparaison avec l'algorithme récent de Moraga, DePuy et Whitehouse [MDW05] (colonnes *Ra-Ps*). Nous mentionnons finalement les résultats obtenus par CPLEX (colonnes *CPLEX*) avec un temps similaire à notre approche et

par Vasquez et Vimont [VV05] (colonnes *VV*) pour les instances comportant 500 variables. Comme les solutions optimales de toutes ces instances ne sont pas connues, nous donnons pour chaque algorithme la déviation moyenne par rapport à la valeur optimale de la relaxation en continu. Dans toute la suite les noms des populations sont identiques à ceux de la section 4.5.1 précédente.

4.5.2.1 Recherche dispersée initiale

Le tableau 4.5 synthétise les résultats obtenus par l'algorithme de recherche dispersée utilisé sans le générateur de solutions basé sur la programmation linéaire proposé dans la section 4.2.3. Une population initiale est déterminée à l'aide du générateur dichotomique. Nous donnons les résultats pour les populations suivantes : *MDNr_SP* (utilisant la méthode chemin1) et *MDNr_SP*2, *MD_SP*2 utilisant la méthode chemin2. Le nombre de solutions de chaque type dans les populations élites et diverses (*MD*) et dans les populations élites, diverses et non réalisables (*MDNr*) est fixé selon le tableau 4.4 suivant dans lequel *M* (resp. *D*, *Nr*) désigne les solutions élites (resp. diverses, non réalisables). Nous avons 40 solutions pour $n = 100$ et 20 solutions pour $n = 250$ et 500.

	n	100	250-500
MD	*M*	15	5
	D	25	15
MDNr	*M*	15	5
	D	20	10
	Nr	5	5

Tableau 4.4 – Fixation de la taille des populations.

Nous précisons dans le tableau 4.5 pour chaque classe d'instances le nombre d'itérations de la recherche dispersée (*nb_iter*), puis la déviation moyenne par rapport à la valeur optimale de la relaxation en continu pour chaque algorithme. La colonne *CPU* donne le temps d'exécution moyen de nos algorithmes (en secondes).

Avant de commenter le tableau 4.5, nous apportons quelques précisions sur certains résultats non mentionnés dans ce tableau. Nous avons testé une version de l'algorithme utilisant les méthodes de chemins comme combinaison. Il s'avère que le processus obtenu est beaucoup plus coûteux et que le temps moyen sur les instances de taille $n = 500$ et $m = 5$ est de l'ordre de 2000 secondes. Ces versions ne sont donc pas très utilisables en pratique, d'autant plus que la qualité de la meilleure solution finale est moins bonne que celle d'autres algorithmes présentés dans le tableau 4.5. Nous avons également testé une version de l'algorithme utilisant une stratégie dynamique pour la mise à jour de la population pendant les

combinaisons. Il s'avère que le temps d'exécution est généralement plus faible mais que la qualité des solutions est en moyenne un peu moins bonne, contrairement aux résultats précédents.

			Chemin1	Chemin2		*CPU*	*VV*	*CPLEX*	*CB*	*HV*	*Ra-Ps*
n	*m*	*nb_iter*	*MDNr_SP*	*MDNr_SP2*	*MD_SP2*						
100		5*n*	0.75	0.59	0.59	21	*N/A*	0.58	0.58	0.72	0.60
250	5	5*n*	0.85	0.19	0.18	54	*N/A*	0.14	0.14	0.36	0.17
500		5*n*	0.87	0.08	0.08	422	0.04	0.05	0.05	0.34	0.09
100		5*n*	1.36	0.96	0.98	26	*N/A*	0.94	0.94	1.26	1.17
250	10	5*n*	0.94	0.38	0.39	69	*N/A*	0.29	0.30	0.74	0.45
500		3*n*	0.83	0.21	0.21	341	0.10	0.12	0.13	0.64	0.20
100		5*n*	2.30	1.71	1.71	39	*N/A*	1.70	1.69	2.14	2.23
250	30	4*n*	1.61	0.82	0.83	87	*N/A*	0.64	0.67	1.36	1.38
500		2*n*	1.67	0.55	0.56	342	0.28	0.33	0.35	1.20	0.82

Tableau 4.5 – Résultats de la recherche dispersée seule.

Les résultats mentionnés dans le tableau 4.5 montrent que l'utilisation de la méthode chemin2 comme processus d'intensification permet d'obtenir de meilleures solutions que la méthode chemin1. En terme de temps d'exécution cela a un coût supplémentaire mais le temps moyen maximum de l'algorithme *MDNr_SP*2 est de l'ordre de 7 minutes (pour $n = 500$ et $m = 5$). Le fait de fixer le nombre d'itérations en fonction de la taille des instances nous permet de limiter l'augmentation du temps total de l'algorithme. Les deux versions utilisant la méthode chemin2 obtiennent des résultats très proches. L'apport des solutions non réalisables n'apparaît pas évident ici : une version peut dominer l'autre sur une classe d'instances et être dominée sur une autre classe. Un point délicat de l'algorithme est le réglage du nombre de solutions de chaque type à l'intérieur de la population référence. Le fait de modifier les valeurs du tableau 4.4 peut en effet changer en partie les résultats du tableau 4.5. Notons que la qualité moyenne de la meilleure solution est meilleure que celle des algorithmes de Haul et Voss et celle de Moraga, DePuy et Whitehouse (excepté pour $n = 500$ et $m = 10$). L'algorithme génétique de Chu et Beasley domine cependant nos deux meilleures versions. Finalement les valeurs obtenues par Vasquez et Vimont sont nettement supérieures, et celles de CPLEX sont un peu meilleures que celles de Chu et Beasley.

4.5.2.2 Population initiale élite

Nous présentons dans le tableau 4.6 les résultats obtenus lorsque nous utilisons notre générateur de solutions basé sur la programmation linéaire pour obtenir une population initiale. Nous précisons le nombre d'itérations effectuées dans l'algorithme de recherche dispersée dans la colonne *nb_iter* pour chaque valeur de *n* et *m*. La colonne *phase initiale*

précise la déviation moyenne de la meilleure solution obtenue par notre générateur utilisé seul et la colonne suivante (*CPU*) donne le temps d'exécution correspondant en secondes. Nous donnons ensuite la déviation moyenne obtenue avec la population *MDNr_SP*2, puis celle avec la même population mais en augmentant sa taille (50 solutions au lieu de 40 pour $n = 100$ et 30 au lieu de 20 pour $n > 100$) (colonne *MDNr_SP*2 *étendue*), et enfin avec une population *MD_SP*2 plus grande également (colonne *MD_SP*2 *étendue*). La colonne *CPU* suivante donne le temps d'exécution moyen de l'algorithme avec une plus grande population.

n	*m*	*nb_iter*	*phase initiale*	*CPU*	*MDNr_SP2*	*MDNr_SP2 étendue*	*MD_SP2 étendue*	*CPU*	*VV*	*CPLEX*	*CB*	*HV*	*Ra-Ps*
100		5**n*	0.66	1	0.59	0.58	0.58	24	*N/A*	0.58	0.58	0.72	0.60
250	5	5**n*	0.18	4	0.16	0.15	0.15	48	*N/A*	0.14	0.14	0.36	0.17
500		5**n*	0.06	60	0.06	0.05	0.05	145	0.04	0.05	0.05	0.34	0.09
100		5**n*	1.10	1	0.96	0.94	0.95	44	*N/A*	0.94	0.94	1.26	1.17
250	10	5**n*	0.36	6	0.33	0.31	0.31	189	*N/A*	0.29	0.30	0.74	0.45
500		3**n*	0.15	81	0.15	0.13	0.15	791	0.10	0.12	0.13	0.64	0.20
100		5**n*	1.73	15	1.69	1.68	1.68	66	*N/A*	1.70	1.69	2.14	2.23
250	30	4**n*	0.68	86	0.67	0.66	0.66	222	*N/A*	0.64	0.67	1.36	1.38
500		2**n*	0.34	430	0.34	0.33	0.34	908	0.28	0.33	0.35	1.20	0.82

Tableau 4.6 – Résultats obtenus avec la population initiale élite.

Nous pouvons tout d'abord remarquer dans le tableau 4.6 que la qualité des solutions initiales obtenues par notre générateur est intéressante. Toutes les moyennes sont meilleures que celles de l'algorithme génétique de Haul et Voss, et une partie domine aussi les résultats de l'algorithme de Moraga, DePuy et Whitehouse. Le temps d'exécution associé varie entre 1 et 90 secondes sur l'ensemble des instances, excepté pour celles de tailles $n = 500$ et $m = 30$ pour lesquelles la moyenne est de l'ordre de 7 minutes. Le processus obtenu est donc assez rapide et permet de générer une population initiale élite. Les résultats des trois colonnes suivantes sont à nouveau assez proches en moyenne. L'impact des solutions non réalisables sur la qualité de la solution finale se voit un peu plus que précédemment en particulier pour les instances comprenant 500 variables pour lesquelles la version *MDNr_SP*2 *étendue* domine la version *MD_SP*2 *étendue*. Notons que la qualité moyenne des solutions est ici du même ordre que celle de l'algorithme de génétique de Chu et Beasley (-0.001% en moyenne), et surtout que les deux dernières versions améliorent les résultats de cet algorithme sur les plus grandes instances comportant 30 contraintes. Notre algorithme obtient des solutions finales proches et parfois meilleures que celles de CPLEX (en moyenne à 0.01%), et les solutions de Vasquez et Vimont restent supérieures sur les instances avec $n = 500$.

Le temps d'exécution de l'algorithme *MDNr_SP*2 varie entre 15 et 350 secondes, alors que pour les versions comportant plus de solutions il est compris entre 25 et 900 secondes en moyenne. Le temps total de l'algorithme (phase initiale + recherche dispersée) est ainsi compris entre 25 secondes et 22 minutes. Il est difficile de faire une comparaison en terme de temps d'exécution entre nos algorithmes et les algorithmes de la littérature en raison des différences entre les machines utilisées. Notons toutefois que l'approche récente de Moraga, DePuy et Whitehouse demande pour ces instances un temps moyen compris entre 7 et 35 minutes sur un PentiumIV 1,6GHz. L'algorithme de Vasquez et Vimont requiert entre 50 et 100 heures par instance sur un PentiumIV 2GHz.

4.6 Conclusions

Nous avons proposé dans ce chapitre un algorithme basé sur la recherche dispersée pour résoudre le problème du sac-à-dos multidimensionnel. Cette approche est une métaheuristique ayant permis l'obtention de bons résultats sur plusieurs problèmes d'optimisation difficiles. Différents travaux basés sur des approches évolutionnaires référencent des résultats intéressants pour le problème du sac-à-dos multidimensionnel, mais il n'existe pas à notre connaissance de travaux récents basés sur la recherche dispersée présentant des résultats numériques pour ce problème. Après une présentation de quelques références bibliographiques nous avons passé en revue les composants majeurs de la méthode. Nous avons proposé et comparé dans un premier temps l'utilisation de plusieurs générateurs de solutions diverses. Nous avons aussi proposé un nouveau générateur de solutions permettant d'obtenir une population initiale élite. Nous avons ensuite évalué expérimentalement le comportement de l'algorithme en fonction de modifications apportées à plusieurs de ses composants. Parmi ceux-ci nous avons insisté plus particulièrement sur la décomposition de la population en sous-populations et l'utilisation de plusieurs méthodes de combinaison. Nous avons montré que l'utilisation de solutions non réalisables au sein de la population peut être bénéfique lors de la phase de recherche dispersée pour ce problème. Nous avons également intégré des phases d'intensification et l'utilisation de la mémoire pour mieux guider la recherche et améliorer les résultats obtenus. L'algorithme combine ainsi des éléments de la recherche tabou et de la méthode des chemins reliants avec la recherche dispersée. Les expériences numériques réalisées sur un ensemble de 270 instances difficiles et corrélées de problèmes de sac-à-dos multidimensionnel montrent que cet algorithme est rapide et assez efficace. L'influence de la population initiale sur le comportement de la

recherche dispersée est également clairement démontrée. L'utilisation de notre générateur pour obtenir une population initiale élite permet ainsi d'obtenir des résultats proches en moyenne de ceux d'un algorithme génétique efficace pour le problème du sac-à-dos multidimensionnel.

Chapitre 5

Heuristiques convergentes basées sur des relaxations

Nous présentons dans ce chapitre plusieurs méthodes de résolution pour les problèmes en variables 0-1 permettant de générer des bornes inférieures et supérieures de bonne qualité. Ces algorithmes reposent sur des relaxations, et ils convergent vers une solution optimale du problème. Certains concepts reposent sur un algorithme proposé à la fin des années 1970. Une des idées est qu'une solution optimale de la relaxation en continu est souvent proche d'une solution optimale du problème. Le principe de l'algorithme est de résoudre de manière exacte une série de problèmes réduits générés à partir d'une série de relaxations en continu. Les résultats obtenus sur différents problèmes de décision sont intéressants, même si la convergence théorique de l'approche est difficilement utilisable en pratique.

Nous commençons ce chapitre par une nouvelle présentation de cet algorithme dans laquelle nous incluons plusieurs propriétés dont une nouvelle preuve de la convergence. Nous proposons ensuite d'inclure différentes améliorations dans cette approche. Celle-ci est alors utilisée de manière heuristique avec un nombre fixé d'itérations. Nous mettons en particulier en évidence deux propriétés de dominance permettant de réduire le nombre de problèmes réduits à résoudre, et donc de réduire théoriquement le temps total d'exécution de l'algorithme. Nous mettons également en œuvre quelques techniques classiques permettant de réduire le temps d'exécution nécessaire à la résolution des problèmes réduits. Devant les résultats prometteurs obtenus, nous proposons ensuite deux nouvelles heuristiques basées sur l'utilisation conjointe de la relaxation en continu et de la relaxation en nombres entiers mixtes du problème. L'intégration de la relaxation en nombres entiers mixtes permet d'ajouter de la diversité dans l'approche. Nous améliorons également dans la majorité des cas la qualité des bornes inférieures et supérieures obtenues. Les algorithmes décrits dans ce chapitre combinent à la fois des méthodes de résolution exacte et approchée. Ils sont applicables aux problèmes en variables 0-1 et sont plus efficaces lorsque le nombre de contraintes est petit par rapport au nombre de variables. L'application de ces algorithmes sur le problème du sac-à-dos

multidimensionnel permet d'améliorer 14 meilleures solutions connues sur un ensemble de 108 problèmes corrélés et difficiles disponibles sur Internet.

5.1 Heuristique itérative basée sur la programmation linéaire

5.1.1 Heuristique basée sur la programmation linéaire

L'algorithme initial basé sur la programmation linéaire a été proposé par Soyster, Lev et Slivka [SLS78] à la fin des années 1970. Il s'agit d'un algorithme exact pour les problèmes en variables 0-1 qui consiste à résoudre une série de sous-problèmes de petite taille générés à partir d'une série de relaxations en continu. Dans toute la suite de ce chapitre nous parlons de problème réduit pour désigner ces sous-problèmes, et nous nous plaçons dans le cadre de la résolution d'un problème P de la forme (1.1) (0-1*PE*).

Rappelons tout d'abord la définition d'un problème réduit $P(x^0,J)$ obtenu à partir d'une solution réalisable du problème P, x^0, et d'un sous ensemble de variables $J \subseteq N$.

$$P(x^0,J) \quad \begin{cases} \max & cx \\ s.c. & Ax \le b \\ & x_j = x_j^0 \quad \forall j \in J \\ & x_j \in \{0,1\} \quad \forall j \in N \end{cases}$$

Rappelons aussi que $v(P)$ dénote la valeur optimale du problème P.

Proposition 5.1 :

Soit x^0 une solution réalisable du problème P et soient J et J' deux sous-ensembles de N avec $J \subseteq J' \subseteq N$. Les inégalités suivantes sont alors respectées.

$$v\left(P\left(x^0, J'\right)\right) \le v\left(P\left(x^0, J\right)\right) \le v(P)$$

Preuve :

La première inégalité est immédiate car le problème $P(x^0,J)$ est une relaxation du problème $P(x^0,J')$. La seconde inégalité est une conséquence de la première puisque $P = P(x^0,\varnothing)$. □

Dans la suite de ce chapitre nous utilisons les notations suivantes. Soit $x \in [0,1]^n$ un vecteur de dimension n, nous confondons régulièrement indices des variables et variables.

L'ensemble des indices des variables fractionnaires de la solution x est dénoté $J^*(x)$

$$J^*(x) = \{j \in N : x_j \in]0,1[\}.$$

L'ensemble des indices des variables entières (resp. égales à 0, resp. égales à 1) dans la solution x est défini par

$$J(x) = \{j \in N : x_j \in \{0,1\}\}$$

(resp. $J^0(x) = \{j \in N : x_j = 0\}$, resp. $J^1(x) = \{j \in N : x_j = 1\}$) ($J(x) = J^0(x) \cup J^1(x)$).

Nous notons aussi par $F(P)$ l'ensemble des solutions réalisables du problème P.

Finalement, soit Q un ensemble de contraintes, nous notons $(P|Q)$ le problème d'optimisation obtenu en ajoutant les contraintes Q au problème P.

L'algorithme proposé par Soyster, Lev et Slivka [SLS78] repose sur une heuristique basée sur la programmation linéaire (*HPL*). Celle-ci génère une borne inférieure et une borne supérieure du problème. Dans la première étape de *HPL* nous résolvons la relaxation en continu du problème pour en obtenir une solution optimale $\bar{x}$. Le problème réduit généré en ne conservant que les variables fractionnaires de la solution $\bar{x}$ comme variables libres (i.e. le problème $P(\bar{x},J(\bar{x})))$ est ensuite résolu de manière exacte pour générer une solution réalisable de P.

Le théorème suivant (Kellerer, Pferschy, et Pisinger [KPP04 chap.9.2] par exemple) permet de connaître une borne supérieure sur la taille maximale du problème réduit $P(\bar{x},J(\bar{x}))$.

Théorème 5.1 :

Toute solution réalisable de la relaxation en continu d'un problème d'optimisation P comportant m contraintes ($m \leq n$) a au plus m variables fractionnaires.

L'heuristique *HPL* est donc plus efficace lorsque le nombre de contraintes du problème est petit par rapport au nombre de variables. Si la taille du problème réduit est importante ou si sa résolution devient trop difficile il peut alors être résolu de manière approchée à l'aide d'une heuristique efficace. Nous utilisons dans nos expériences numériques une méthode du simplexe pour résoudre la relaxation en continu du problème et une méthode de séparation et évaluations pour obtenir une solution optimale du problème réduit (à l'aide du logiciel CPLEX d'Ilog). L'algorithme 5.1 donne une description de l'heuristique *HPL*.

Algorithme 5.1 Heuristique basée sur la programmation linéaire

Entrée : le problème *P*.

Sorties : une solution réalisable du problème x^0; une solution optimale de la relaxation en continu $\bar{x}$.

1 : Résoudre la relaxation en continu de *P* et récupérer une solution optimale $\bar{x}$.

2 : Résoudre le problème réduit $P(\bar{x}, J(\bar{x}))$ pour obtenir une solution optimale x^0.

3 : Retourner la solution réalisable x^0 de *P* et la solution optimale $\bar{x}$ de *LP*(*P*).

Notons que lorsque la relaxation en continu *LP*(*P*) du problème *P* n'est pas réalisable alors le problème *P* lui-même est non réalisable. De même si une solution optimale de *LP*(*P*) est entière alors elle est optimale pour le problème *P*. Cette heuristique est utilisable pour le problème *SADM*. Il suffit d'appliquer $HPL(SADM, x^0, \bar{x})$.

Nous avons parlé dans le chapitre 2 d'heuristiques d'arrondi de la relaxation en continu. L'heuristique *HPL* permet souvent d'obtenir des solutions réalisables de meilleures qualités que celles-ci. Nous illustrons cela dans les deux tableaux 5.1 et 5.2.

Problème	***n***	***m***	***Opt***	***Tronc***	***Arr***	***HPL***
Pet1	6	10	3800	3200	3200	**3700**
Pet3	15	10	4015	2805	2805	**3105**
Pet4	20	10	6120	5600	5920	5920
Pet5	28	10	12400	11140	11140	**11720**
Pet6	39	5	10618	9532	10278	**10479**
Pet7	50	5	16537	16144	16235	16235
flei	20	10	*2139*	*2139*	*2139*	*2139*
hp1	28	4	3418	2332	2447	**3279**
hp2	35	4	3186	2793	2884	2884
fp1	27	4	3090	2004	2119	**2951**
fp2	34	4	3186	2788	2788	**2884**
fp4	29	2	95168	66929	66929	66929
fp5	20	10	2139	1235	2025	**2045**
fp6	40	30	776	538	643	**738**
fp7	37	30	*1035*	898	898	***1035***
sento1	60	30	7772	7534	7639	**7734**
sento2	60	30	*8722*	8585	8585	***8722***
weing1	28	2	141278	139508	139508	139508
weing2	28	2	130883	129613	129613	**129723**
weing3	28	2	95677	72010	94517	94517
weing4	28	2	119337	103939	103939	**104689**
weing5	28	2	98796	83868	83868	83868
weing6	28	2	130623	129723	129723	129723
weing7	105	2	1095445	1093552	1093552	1093552
weing8	105	2	624319	596080	596080	596080

Tableau 5.1 – Heuristiques basées sur la programmation linéaire et instances classiques.

Nous comparons la valeur de la borne inférieure obtenue en appliquant une heuristique de troncature (*Tronc*), une heuristique d'arrondi suivie d'une heuristique gloutonne pour

obtenir une solution réalisable si nécessaire (*Arr*) et enfin *HPL*. Nous précisons aussi la valeur optimale du problème, *Opt*. Les tests ont été effectués sur 55 instances de problèmes de *SADM* de petites tailles (*n* allant de 6 à 105 et *m* de 2 à 30) décrites dans le chapitre 4 (section 4.2.2.3). Nous donnons pour chaque instance son nom et sa taille avec les valeurs de *n* et *m*.

Problème	***n***	***m***	***Opt***	***Tronc***	***Arr***	***HPL***
weish1	30	5	4554	4466	4466	4466
weish2	30	5	4536	4414	4414	4414
weish3	30	5	4115	3783	3842	**4106**
weish4	30	5	*4561*	4468	*4561*	*4561*
weish5	30	5	4514	4460	4460	4460
weish6	40	5	5557	5413	5533	5533
weish7	40	5	5567	5464	5464	5464
weish8	40	5	5605	5481	5481	**5517**
weish9	40	5	*5246*	*5246*	*5246*	*5246*
weish10	50	5	6339	6311	6311	6311
weish11	50	5	*5643*	*5643*	*5643*	*5643*
weish12	50	5	6339	6311	6311	6311
weish13	50	5	6159	6074	6133	6133
weish14	60	5	6954	6914	6914	6914
weish15	60	5	7486	7458	7458	7458
weish16	60	5	7289	7162	7162	7162
weish17	60	5	8633	8586	8586	8586
weish18	70	5	9580	9489	9489	9489
weish19	70	5	7698	7588	7588	7588
weish20	70	5	9540	9393	9393	9393
weish21	70	5	9074	8818	8818	8818
weish22	80	5	8947	8628	8687	8687
weish23	80	5	8344	8259	8259	8259
weish24	80	5	10220	10142	10142	**10170**
weish25	80	5	9939	9776	9776	9776
weish26	90	5	9584	9492	9492	9492
weish27	90	5	9819	9720	9779	9779
weish28	90	5	*9492*	*9492*	*9492*	*9492*
weish29	90	5	*9410*	8978	*9410*	*9410*
weish30	90	5	11191	11131	11131	11131

Tableau 5.2 – Heuristiques basées sur la programmation linéaire et instances classiques (fin).

Les valeurs en gras indiquent les instances pour lesquelles une heuristique domine les deux autres. L'heuristique *HPL* obtient 17 fois une meilleure solution que les heuristiques d'arrondi et elle n'est surtout jamais dominée par les deux autres. Nous pouvons obtenir 8 solutions optimales avec ces heuristiques (les valeurs associées sont en italique). Notons que pour les instances du tableau 5.2 les heuristiques de troncature et d'arrondi sont assez proches de *HPL* et les trois heuristiques permettent d'obtenir des bornes inférieures proches des valeurs optimales des instances. La résolution de ces problèmes ne pose aujourd'hui plus de problèmes aux logiciels d'optimisation et les temps d'exécution des heuristiques sont faibles et du même ordre.

Même si *HPL* semble être une heuristique performante et très rapide pour les problèmes peu contraints, elle ne permet pas toujours de générer une solution réalisable pour tout problème d'optimisation en variables 0-1. Cela est en particulier le cas lorsqu'il existe des coefficients négatifs dans les données du problème. Cette situation peut aussi apparaître lors de la résolution de problèmes particuliers comme le problème du sac-à-dos multidimensionnel avec contraintes de demande (*SADMD*) comme nous l'illustrons dans l'exemple suivant.

Exemple 5.1 : Soit l'instance suivante de problème de *SADMD* (*P*1) comportant 13 variables, 5 contraintes de capacité et 1 contrainte de demande.

c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	
41	36	36	36	27	30	53	34	35	30	51	42	36	
A^1	A^2	A^3	A^4	A^5	A^6	A^7	A^8	A^9	A^{10}	A^{11}	A^{12}	A^{13}	b_i
12	22	22	87	7	55	14	23	88	16	95	99	28	313(≤)
29	95	78	6	85	25	76	29	79	64	80	55	48	403(≤)
74	8	44	73	97	75	32	47	15	14	33	11	17	240(≤)
64	21	66	2	73	98	52	91	27	82	99	32	99	415(≤)
93	92	65	46	4	33	65	70	9	45	26	46	91	338(≤)
96	49	18	36	4	95	62	34	50	37	22	21	74	306(≥)

La solution $\bar{x}$ suivante est une solution optimale de la relaxation en continu de (*P*1)

$$\bar{x} = (1\ 0.18\ 0\ 0.22\ 0.24\ 0\ 1\ 0.58\ 0.72\ 1\ 1\ 0.76\ 0).$$

Sa valeur est 272.34. Le problème réduit associé à $\bar{x}$ est obtenu en conservant uniquement les variables $x_2, x_4, x_5, x_8, x_9, x_{12}$.

$$P1(\bar{x},J(\bar{x})) \begin{cases} \max & 36x_2 + & 36x_4 + & 27x_5 + & 34x_8 + & 35x_9 + & 42x_{12} & \\ & 22x_2 + & 87x_4 + & 7x_5 + & 23x_8 + & 88x_9 + & 99x_{12} & \le 176 \\ & 95x_2 + & 6x_4 + & 85x_5 + & 29x_8 + & 79x_9 + & 55x_{12} & \le 154 \\ & 8x_2 + & 73x_4 + & 97x_5 + & 47x_8 + & 15x_9 + & 11x_{12} & \le 87 \\ & 21x_2 + & 2x_4 + & 73x_5 + & 91x_8 + & 27x_9 + & 32x_{12} & \le 118 \\ & 92x_2 + & 46x_4 + & 4x_5 + & 70x_8 + & 9x_9 + & 46x_{12} & \le 109 \\ & 49x_2 + & 36x_4 + & 4x_5 + & 34x_8 + & 50x_9 + & 21x_{12} & \ge 89 \\ & x_j \in \{0,1\} & j = 2,4,5, & 8,9,12 & & & & \end{cases}$$

Il est facile de vérifier que le problème réduit $P1(\bar{x},J(\bar{x}))$ n'a pas de solution réalisable alors que le problème (*P*1) est réalisable.

La solution $x^* = (1\ 1\ 0\ 0\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 0\ 0)$ est en effet une solution optimale de (*P*1) avec une valeur de 246. Même si *HPL* n'est pas capable de générer de solution réalisable pour tout problème en nombres entiers elle est par contre appropriée pour le problème *SADM*.

Glover [Glo05] a récemment proposé une méthode de résolution (*AMP*) pour les problèmes en nombres entiers mixtes qui combine des techniques de projection et des processus de mémoire adaptive de la recherche tabou. Cette approche est particulièrement efficace à l'intérieur de métaheuristiques pour la résolution de problèmes d'optimisation de grande taille. Il montre que l'intégration de processus d'intensification et de diversification dans les méthodes de projection peut être enrichi en incluant des pseudo-coupes qui guident la recherche. L'idée de *AMP* consiste à générer des solutions en laissant itérativement des sous-ensembles de variables fixées à une valeur pendant que les valeurs des autres variables changent. De nouvelles contraintes sont généralement ajoutées dans le modèle pour fixer une partie des variables à des valeurs particulières. Cette approche englobe différentes méthodes comme la recherche dans un voisinage large (*LNS*) introduite par Shaw [Sha98], la recherche dans un voisinage large étendu proposée par Ahuja et al. [AEOP02], la méthode de branchement local proposée par Fischetti et Lodi [FL03], la recherche dans des voisinages induits par des relaxations (*RINS*) de Danna, Rothberg et Pape [DRP05]. *LNS* et *RINS* ont été appliquées avec succès pour la résolution de problèmes d'optimisation en nombres entiers mixtes de grandes tailles. L'algorithme *RNIS* résout des problèmes réduits à chaque nœud de l'arbre de recherche d'un algorithme de séparation et coupes. Plus précisément, en notant x^0 la meilleure solution courante réalisable et $\bar{x}$ la solution de la relaxation en continu au nœud courant de l'arbre de recherche, *RINS* résout le problème réduit suivant

$$(P((\bar{x}+x^0)/2, J((\bar{x}+x^0))/2) | cx > cx^0).$$

L'heuristique *HPL* permet d'obtenir un encadrement $[cx^0; c\bar{x}]$ de la valeur optimale du problème initial. L'algorithme proposé par Soyster, Lev et Slivka [SLS78] que nous appelons Heuristique Itérative basée sur la Programmation Linéaire dans la suite repose sur *HPL* et réduit cet encadrement jusqu'à obtenir une solution optimale du problème.

5.1.2 Heuristique itérative basée sur la programmation linéaire

Le principe de l'heuristique itérative basée sur la programmation linéaire (*HIPL*) est d'appliquer l'heuristique *HPL* itérativement sur le problème *P*. Celui-ci change à chaque itération pour modifier le sous-espace de recherche exploré de manière optimale lors de la résolution exacte du problème réduit. Pour cela, une contrainte est ajoutée au problème pour diminuer la valeur de la borne supérieure à l'itération suivante, tout en dirigeant la recherche vers une région inexplorée. La contrainte ajoutée à chaque itération repose sur la proposition suivante.

Proposition 5.2 : soit x^0 un vecteur dans $\{0, 1\}^n$, l'inégalité suivante

$$\sum_{j\in J^1(x^0)} x_j - \sum_{j\in J^0(x^0)} x_j \le \left|J^1(x^0)\right| - 1 \qquad (5.1)$$

élimine la solution x^0 sans éliminer d'autres solution dans $\{0,1\}^n$.

Preuve : pour toute solution $x \neq x^0$ on a

$$\| x^0 - x\|_1 > 0 \qquad (5.1\text{-a})$$

En utilisant la définition de la norme $\| \cdot \|_1$, l'inégalité stricte (5.1-a) peut alors s'exprimer comme suit

$$\sum_{j\in N} \left|x_j^0 - x_j\right| > 0 .$$

Comme les deux vecteurs sont entiers, on a

$$\sum_{j\in J^1(x^0)} \left|x_j^0 - x_j\right| + \sum_{j\in J^0(x^0)} \left|x_j^0 - x_j\right| > 0 \qquad (5.1\text{-b})$$

L'inégalité (5.1-b) peut alors être développée comme suit

$$\sum_{j\in J^1(x^0)} \left|1 - x_j\right| + \sum_{j\in J^0(x^0)} \left|x_j\right| > 0 .$$

Et en développant à nouveau nous obtenons l'inégalité (5.1-c)

$$\left|J^1\left(x^0\right)\right| - \sum_{j\in J^1\left(x^0\right)} x_j + \sum_{j\in J^0\left(x^0\right)} x_j > 0 \qquad (5.1\text{-c})$$

Comme les solutions x sont binaires et les données sont entières, l'inégalité (5.1-c) est équivalente à (5.1). □

L'algorithme *HIPL* utilise des contraintes de la forme (5.1) pour éliminer de l'espace de recherche les solutions réalisables préalablement visitées lors de la résolution des problèmes réduits. Pour cela la contrainte est générée selon une solution optimale $\bar{x}$ de la relaxation en continu du problème *P*. Chaque variable égale à 1 (resp.0) dans la solution courante $\bar{x}$ reçoit un coefficient égal à 1 (resp. -1) dans la contrainte. Les autres variables (i.e. les variables fractionnaires) ont un coefficient égal à 0. De manière à éliminer l'ensemble des solutions du problème réduit courant $P(\bar{x},J(\bar{x}))$, la somme des variables sur *N* doit être inférieure au nombre de variables égales à 1 dans $\bar{x}$ (i.e. $|J^1(\bar{x})|$). Plus formellement la contrainte est générée selon la proposition suivante.

Proposition 5.3 : Soient un problème d'optimisation *P*, une solution optimale $\bar{x}$ de la relaxation en continu de *P*, et une solution optimale x^0 du problème réduit $P(\bar{x},J(\bar{x}))$.

Une solution optimale de *P* est soit la solution x^0, soit une solution optimale du problème suivant

$$(P \mid \{fx \leq |J^1(\bar{x})|\text{-}1\}) \tag{5.2-a}$$

avec *f* le vecteur de dimension *n* défini par

$$f_j = \begin{cases} 1 & \text{si } \bar{x}_j = 1 \\ -1 & \text{si } \bar{x}_j = 0 \\ 0 & \text{si } \bar{x}_j \in]0,1[\end{cases} \tag{5.2-b}$$

Preuve : Il est facile de voir que pour chaque solution *x* dans $\{0,1\}^n$ on a

$$fx \leq |J^1(\bar{x})|.$$

Cette inégalité peut être divisée en deux inégalités pour chaque solution *x* du problème *P* car toutes les données sont entières

$$fx < |J^1(\bar{x})| \tag{5.2-c}$$

ou

$$fx = |J^1(\bar{x})| \tag{5.2-d}$$

Chaque solution vérifiant (5.2-d) est considérée dans le problème réduit $P(\bar{x},J(\bar{x}))$ et peut donc être éliminée à l'itération suivante d'après la proposition 5.2. Comme les variables sont binaires et que les coefficients dans la contrainte (5.2-c) sont entiers, cette contrainte peut être reformulée comme suit

$$fx \leq |J^1(\bar{x})|\text{-}1 \tag{5.3}$$

Ainsi une solution optimale de *P* ne peut pas être éliminée par (5.2-c). □

Les contraintes ajoutées dans le problème (5.2-a) sont appelées coupes canoniques sur l'hypercube unité par Balas et Jeroslow [BJ72]. L'exemple suivant illustre la génération d'une contrainte de type (5.3).

Exemple 5.2 : Soit l'instance de problème de sac-à-dos multidimensionnel suivante.

$$(Pb)\quad \begin{cases} \max & 14x_1 + & 10x_2 + & 8x_3 + & 7x_4 + & 4x_5 & \\ s.c. & 10x_1 + & 8x_2 + & 7x_3 + & 5x_4 + & 2x_5 & \le 20 \\ & 9x_1 + & 11x_2 + & 5x_3 + & 7x_4 + & 4x_5 & \le 20 \\ & 7x_1 + & 7x_2 + & 4x_3 + & 2x_4 + & 4x_5 & \le 15 \\ & x_j \in \{0,1\} & & j = 1,...,5 & & & \end{cases}$$

Une solution optimale de la relaxation en continu de ce problème est définie par

$$\bar{x} = (1 \;\; 0 \;\; 0.86 \;\; 0.43 \;\; 0.93).$$

La contrainte obtenue à partir de (5.2a) est donc $x_1 - x_2 \le 0$ puisque toutes les autres variables sont fractionnaires. Si la contrainte était $x_1 - x_2 = 1$ les seules solutions réalisables du problème à l'itération suivante seraient celles vérifiant $x_1 = 1$ et $x_2 = 0$. Ces solutions sont celles qui correspondent au problème réduit obtenu en laissant les variables x_3, x_4 et x_5 libres. Or ce problème réduit est résolu de manière optimale à l'itération courante. L'ajout de la contrainte (5.3) dans le problème *P* permet donc d'éliminer définitivement ces solutions.

Nous décrivons dans l'algorithme 5.2 l'heuristique *HIPL*. Le processus s'arrête lorsque la différence entre la valeur de la borne supérieure (i.e. la valeur de $LP(P)$) et la valeur de la borne inférieure (i.e. la meilleure solution réalisable) devient inférieure à 1. Si cette condition est vérifiée (ligne 8) alors la solution correspondant à la borne inférieure est une solution optimale du problème initial. A chaque itération de l'algorithme une pseudo-coupe générée selon la proposition 5.3 est ajoutée dans le problème (ligne 7).

Algorithme 5.2 Heuristique itérative basée sur la programmation linéaire

Entrée : le problème *P*.
Sortie : une solution optimale x^* de *P*.

1 : Soit x^* une solution réalisable de *P*;
2 : $v^* = cx^*$; *stop* = *faux* ; $Q = P$;
3 : **tant que** (*stop* = faux) **faire**
4 : soit $\bar{x}$ une solution optimale de $LP(Q)$;
5 : soit x^0 une solution optimale de $P(\bar{x}, J(\bar{x}))$;
6 : **si** $cx^0 > v^*$ **alors** $x^* = x^0$; $v^* = cx^*$;
7 : $Q = (Q \mid \{f x \le |J^1(\bar{x})| - 1\})$;
8 : **si** $\lfloor c\bar{x} - v^* \rfloor$ **alors** *stop* = *vrai*;
9 : **fin tantque**

L'ajout de la contrainte (5.3) à chaque itération permet de réduire la valeur de la relaxation en continu. La proposition 5.4 formalise ce principe.

Proposition 5.4 : soit P^k le problème considéré à l'itération k et soit P^{k+1} le problème obtenu à partir de P^k en ajoutant la contrainte (5.3). On a

$$v\left(LP\left(P^{k+1}\right)\right) \leq v\left(LP\left(P^{k}\right)\right) \tag{5.4}$$

Preuve : L'inégalité (5.4) résulte directement du fait que le problème P^k est une relaxation du problème P^{k+1} (i.e. $F(P^{k+1}) \subseteq F(P^k)$). □

Cette proposition assure que l'algorithme *HIPL* construit une séquence de bornes supérieures non-croissantes. Le théorème suivant établit que lorsque l'algorithme *HIPL* s'arrête la meilleure borne inférieure obtenue correspond à la valeur optimale du problème initial. Nous considérons que la valeur optimale de la relaxation en continu du problème P^k est égale à $-\infty$ si elle n'est pas réalisable.

Théorème 5.2 : Soit P un problème d'optimisation avec toutes ses données entières et positives. La meilleure solution réalisable obtenue par l'algorithme *HIPL* est une solution optimale de P. De plus si une solution optimale de $LP(P^k)$ est entière ou si le problème $LP(P^k)$ est non réalisable alors l'algorithme *HIPL* s'arrête.

Preuve : Soit F^k l'union des ensembles de solutions réalisables des problèmes réduits résolus de manière exacte jusque l'itération k par *HIPL*. Plus spécifiquement,

$$F^k = \bigcup_{i=1}^{k} F(P(\bar{x}^i, J^*(\bar{x}^i))).$$

Notons que l'ensemble F^k et l'ensemble des solutions réalisables du problème P^k forment une partition de l'ensemble des solutions réalisables de P, i.e. $F(P) = F^k \cup F(P^k)$ et $F^k \cap F(P^k) = \varnothing$. Soit x^{*k} la solution finale retournée par *HIPL* lorsqu'il s'arrête à l'itération k. Il est clair que l'on a

$$cx^{*k} = max\{cx : x \in F^k\}.$$

De plus on a

$$v(P) = max\{cx : x \in F(P)\} = max\{cx : x \in F^k \cup F(P^k)\}.$$

Donc, on a :

$$v(P) = max\{cx^{*k}; max\{cx : x \in F(P^k)\}\} \tag{5.5}$$

Comme $v(LP(P^k)) \geq max\{cx : x \in F(P^k)\}$ et comme toutes les données de P sont entières, la condition d'arrêt de l'algorithme *HIPL* ($\lfloor v(LP(P^k)) - cx^{*k} \rfloor < 1$) implique que la solution finale x^{*k} est une solution optimale pour P. Il est également clair selon (5.5) que si une solution

optimale de $LP(P^k)$ est entière ou si le problème $LP(P^k)$ n'est pas réalisable alors la solution x^{*k} est optimale pour P. □

Le théorème suivant énonce la convergence finie de *HIPL*.

Théorème 5.3 : L'algorithme *HIPL* converge vers une solution optimale du problème ou indique que le problème est non réalisable en un nombre fini d'itérations.

Preuve : Nous définissons une solution partielle comme étant un vecteur dans lequel une partie des composantes ne sont pas fixées. Dans la suite de cette preuve nous considérons comme solution partielle des solutions complètes ou incomplètes. Plus précisément, une solution partielle d'ordre k est un vecteur pour lequel exactement $n - k$ variables sont affectées à la valeur 0 ou 1. Il y a 2^n solutions partielles d'ordre n (i.e. solutions dans $\{0,1\}^n$), et il y a une solution partielle d'ordre 0 (i.e. la solution ne comportant aucune variable fixée). Il y a ainsi $C_n^k 2^{n-k}$ solutions partielles d'ordre k. Le nombre total de solutions partielles est donc

$$\sum_{k=0}^{n} C_n^k 2^{n-k} = \sum_{k=0}^{n} C_k^n 1^k \times 2^{n-k} = (1+2)^n = 3^n .$$

Observons tout d'abord qu'à chaque itération l'algorithme *HIPL* génère une solution partielle à partir d'une solution optimale de la relaxation en continu de P. A l'itération suivante cette solution partielle est éliminée par l'ajout de la contrainte (5.3). Puisqu'il y a un nombre fini de solutions partielles, le nombre d'itérations de *HIPL* est donc borné par 3^n. De plus, selon le théorème 5.2, *HIPL* s'arrête dès qu'une solution partielle d'ordre n est trouvée. Ainsi le nombre total d'itérations ne peut excéder $3^n - 2^n$. □

Notons que Soyster, Lev et Slivka proposent dans [SLS78] une autre démonstration de la convergence de l'algorithme *HIPL*. Elle repose sur les faces de dimension k de l'hypercube unité. Rappelons qu'une face de dimension k de l'hypercube unité $K=\{x \in IR^n : 0 \leq x_j \leq 1, j=1, \ldots, n\}$ est une collection de points de K pour lesquels exactement $n - k$ des inégalités $0 \leq x_j \leq 1, j=1, \ldots, n$ sont respectées.

Nous illustrons la convergence de *HIPL* dans le tableau 5.3. Nous appliquons l'algorithme sur deux instances de problèmes de sac-à-dos multidimensionnel existantes. La première, *GK*9, comporte 30 variables et 10 contraintes. La seconde, *OR*-100-5.4, comporte

100 variables et 5 contraintes. Nous précisons dans le tableau 5.3 la valeur de la borne supérieure du problème ($\bar{v}$), la valeur de la borne inférieure ($\underline{v}$), et la taille du problème réduit ($|J^*(\bar{x})|$), pour un ensemble d'itérations (*iter*).

*GK*9				*OR*-100.5.4			
iter	$\bar{v}$	$\underline{v}$	$\lvert J^*(\bar{x})\rvert$	*iter*	$\bar{v}$	$\underline{v}$	$\lvert J^*(\bar{x})\rvert$
1	380.3	336	6	1	23724.1	22554	5
2	379.7	368	7	2	23722.7	22983	5
3	379.6	360	8	3	23720.3	23056	7
4	379.5	368	9	4	23715.5	22606	6
5	378.7	368	10	40	23687.4	23447	18
8	377.9	368	10	41	23687.3	**23534**	20
9	377.6	372	10	42	23686.8	23402	16
10	377.2	368	11	98	23652.3	23497	23
11	376.9	372	13	99	23651.6	23486	23
12	376.5	364	7	100	23651.1	23497	24
13	376.4	**376**	9	292	23534.6	23497	46

Tableau 5.3 – Illustration de la convergence sur les instances *GK*9 et *OR*-100.5.4.

L'instance *GK*9 est résolue de manière optimale en 13 itérations par *HIPL*. Une solution de valeur 376 est générée à l'itération 13, et la valeur de la borne supérieure courante à cette itération vaut 376.4. La condition d'arrêt de l'algorithme *HIPL* est donc vérifiée. La résolution de la seconde instance est déjà beaucoup moins évidente. En effet, une solution optimale de valeur 23534 est obtenue à l'itération 41, mais la différence entre les deux bornes est de l'ordre de 0.5% après 100 itérations. Il faut finalement effectuer 292 itérations pour pouvoir prouver l'optimalité de cette solution. Le temps d'exécution associé est de l'ordre de 400 secondes sur notre machine, alors que l'application d'un algorithme de séparation et évaluations seul permet de générer cette solution et d'en prouver l'optimalité en quelques secondes. Remarquons que la taille des problèmes réduits n'augmente pas nécessairement entre deux itérations. En théorie le nombre de variables fractionnaires dans une solution optimale de la relaxation en continu peut ici augmenter d'une unité à chaque itération. Il arrive souvent en pratique que la taille des problèmes réduits ne change pas entre plusieurs itérations consécutives (ou même qu'elle diminue comme pour les deux instances dans le tableau 5.3).

De manière à être plus efficace, nous proposons dans la suite d'utiliser l'algorithme *HIPL* comme une heuristique en remplaçant le critère d'arrêt (ligne 3 de l'algorithme 5.2) par un nombre total d'itérations. Nous présentons plusieurs méthodes pour accélérer le processus et pour réduire l'écart entre les deux bornes finales dans ces conditions.

5.2 Améliorations des performances de *HIPL*

L'algorithme *HIPL* décrit précédemment consomme clairement la majorité de son temps d'exécution lors des résolutions exactes des problèmes réduits. En effet la résolution de la relaxation en continu et la génération de la contrainte à ajouter au problème ne demandent généralement pas beaucoup d'effort grâce aux progrès des logiciels d'optimisation (même si cela est à relativiser en fonction de la taille des problèmes et du nombre d'itérations effectuées). Ces logiciels (et les algorithmes exacts existants) éprouvent cependant toujours des difficultés pour résoudre de manière optimale des problèmes de taille moyenne. C'est le cas pour le problème *SADM* et pour d'autres problèmes d'optimisation. Comme la taille des problèmes à résoudre de manière optimale augmente au cours de *HIPL*, leur résolution devient de plus en plus difficile. Des problèmes réduits obtenus à partir d'instances de problèmes de *SADM* avec n = 500 et m = 30 sont par exemple difficilement résolus de manière optimale par CPLEX dès la première itération de *HIPL*. Nous proposons dans la section suivante deux propriétés pour diminuer le nombre de problèmes réduits à résoudre de manière optimale et accélérer l'algorithme.

5.2.1 Diminution du nombre de problèmes réduits à résoudre

Les propriétés sont basées sur la définition d'une dominance entre deux solutions fractionnaires.

Définition 5.1 :

Soient x^1 et x^2 deux solutions définies dans $[0,1]^n$. Nous disons que la solution x^1 domine la solution x^2 si

$$J^*(x^2) \subseteq J^*(x^1) \text{ et } J^1(x^1) \subseteq J^1(x^2) \text{ et } J^0(x^1) \subseteq J^0(x^2).$$

La proposition suivante décrit la première propriété de dominance associée à cette définition.

Proposition 5.5 :

Soient x^1 et x^2 deux solutions définies dans $[0,1]^n$. Si la solution x^1 domine la solution x^2 au sens de la définition 5.1 alors on a

$$v(P(x^1,J(x^1))) \geq v(P(x^2,J(x^2))) \tag{5.6}$$

Preuve : Si la solution x^1 domine la solution x^2 au sens de la définition 5.1, les variables entières dans la solution x^1 sont les mêmes dans la solution x^2. On a donc $J(x^1) \subseteq J(x^2)$. Comme on a également $J^*(x^2) \subseteq J^*(x^1)$, l'inégalité (5.6) est obtenue à partir de la proposition 5.1. □

L'implication logique de la proposition précédente est que seuls les problèmes réduits relatifs aux solutions non dominées doivent être résolus de manière exacte. L'intégration de la dominance dans le processus *HIPL* consiste à conserver une liste L formée des éléments non dominés. D'un point de vue algorithmique notons qu'un élément de la liste est une solution de la relaxation en continu et que L peut être triée selon la taille des ensembles fractionnaires des solutions pour accélérer la détection des éléments dominés. Pour détecter si un élément est dominé ou s'il domine d'autres éléments, nous utilisons les deux ensembles $D^+(\bar{x})$ et $D^-(\bar{x})$ associés à une solution $\bar{x} \in [0,1]^n$ et définis comme suit

$$D^+(\bar{x}) = \{y \in L : \bar{x} \text{ domine } y\}.$$

$$D^-(x) = \{y \in L : y \text{ domine } \bar{x}\}.$$

Une solution $\bar{x}$ est ainsi ajoutée dans L si et seulement si $D^-(\bar{x}) = \varnothing$.

L'algorithme 5.3 décrit la nouvelle heuristique obtenue. A chaque itération un parcours de la liste est effectué pour détecter et éliminer les éléments dominés dans L.

L'application de la proposition 5.5 permet de diminuer le nombre de problèmes réduits à résoudre de manière optimale sur la majorité des instances que nous avons traitées. Cette diminution varie en fonction de la taille des problèmes (en moyenne de l'ordre de 30 à 50%, voir les sections 5.4 et 5.5). Nous proposons également une extension de cette proposition.

Proposition 5.6 : Soient x^1 et x^2 deux solutions dans $[0,1]^n$, la solution y définie par :

$$y_j = \begin{cases} 0 & \text{si } j \in J^0(x^1) \cap J^0(x^2) \\ 1 & \text{si } j \in J^1(x^1) \cap J^1(x^2) \\ x_j^1 & \text{si } j \in J^*(x^1) \\ x_j^2 & \text{si } j \in J^*(x^2) \end{cases}$$

domine les solutions x^1 et x^2, et on a

$$v(P(y,J(y))) \geq \max\{(v(P(x^1,J(x^1))),v(P(x^2,J(x^2))))\} \qquad (5.7)$$

Preuve : il est facile de vérifier que la solution y domine les deux solutions x^1 et x^2 au sens de la définition 5.1. L'inégalité (5.7) dérive alors de la proposition 5.5. □

Algorithme 5.3 Heuristique itérative basée sur la programmation linéaire avec dominance

Entrées : le problème P; le nombre d'itérations *Max_Iter*.
Sortie : une solution réalisable x^*.

Soit x^* une solution réalisable de P;
$v^* = cx^*$; $L = \emptyset$; $iter = 1$; $Q = P$;
tant que ($iter \leq max\_iter$) **faire**
    Soit $\bar{x}$ une solution optimale de $LP(Q)$;
    $Q = (Q \mid \{f.x \leq |J^1(\bar{x})| - 1\})$;
    si $D^-(\bar{x}) = \emptyset$ **alors** $L = L + \bar{x} - D^+(\bar{x})$;
    $iter = iter + 1$;
fin tantque
pour toute solution $\bar{x} \in L$ **faire**
    Soit x une solution optimale du problème réduit $P(\bar{x}, J(\bar{x}))$;
    si $cx > v^*$ **alors** $x^* = x$; $v^* = cx$;
fin pour

Dans la pratique il est nécessaire d'utiliser un paramètre pour fixer un nombre de variables fractionnaires limite dans la solution y. L'ajout de ce paramètre permet d'éviter d'avoir des problèmes réduits de taille proche ou égale à n (qui n'auraient pas d'intérêt). Le réglage de ce paramètre est important et se fait de manière expérimentale. Il peut dépendre du nombre total d'itérations et/ou du nombre de contraintes du problème par exemple.

Algorithme 5.4 Détection et élimination des éléments dominés dans la liste

Entrées : la liste L; le paramètre t.
Sortie : la liste L.

elimine = *vrai*;
tant que (*elimine* = *vrai*) **faire**
    elimine = *faux*;
    pour $\bar{x} \in L$ **faire**
        pour $\bar{z} \in L$ **faire**
            si $(|J^*(\bar{x})| + |J^*(\bar{z})| - |J^*(\bar{x}) \cap J^*(\bar{z})| + |J^0(\bar{x}) \cap J^1(\bar{z})| + |J^1(\bar{x}) \cap J^0(\bar{z})| \leq t)$ **alors**
                Construire la solution $\bar{y}$ selon la proposition 5.6;
                $L = L + \bar{y} - \bar{x} - \bar{z}$;
                elimine = *vrai* ;
            fin si
        fin pour
    fin pour
fin tanque

Nous présentons dans l'algorithme 5.4 le principe de détection et d'élimination des éléments dominés en appliquant les propositions 5.5 et 5.6 précédentes. Deux paramètres sont nécessaires : la liste L et la taille maximale autorisée pour un problème réduit, notée t. La ligne 6 permet la détection des éléments qui peuvent être supprimés pour être remplacés par un seul en utilisant la proposition 5.6.

5.2.2 Accélération de la résolution d'un problème réduit

Différents moyens peuvent être mis en œuvre pour accélérer ou limiter le temps d'exécution nécessaire pour explorer de manière optimale le sous-espace de recherche associé à un problème réduit. Dans notre approche, nous ajoutons quatre contraintes dans le problème à résoudre. Un premier groupe comporte deux contraintes qui encadrent la valeur optimale du problème à l'aide d'une borne supérieure et d'une borne inférieure. L'intérêt d'utiliser ces contraintes est qu'elles évoluent toutes les deux au cours de l'algorithme *HIPL*. Nous ajoutons plus formellement les contraintes suivantes

$$\underline{v} \leq cx \leq \overline{v} \tag{5.8}$$

avec $\underline{v}$ la meilleure borne inférieure connue et $\overline{v}$ la meilleure borne supérieure connue.

Les deux autres contraintes imposent deux bornes sur la somme des variables du problème. Glover a proposé ce type de contrainte en 1965 [Glo65]. Fréville et Plateau ont montré que ce genre de contraintes peut améliorer la résolution des problèmes *SADM* [FP93a]. Vasquez et Hao [VH01b] ont aussi utilisé ce genre de contraintes. Plus précisément, nous ajoutons les deux contraintes suivantes

$$\underline{\sigma} \leq \sum_{j \in N} x_j \leq \overline{\sigma} \tag{5.9}$$

avec $\underline{\sigma}$ (resp. $\overline{\sigma}$) une borne inférieure (resp. supérieure) sur la somme des variables du problème. Les contraintes (5.9) sont valides pour le problème P avec des valeurs de $\underline{\sigma}$ et $\overline{\sigma}$ égales aux valeurs optimales des programmes linéaires suivants

$$(\underline{P}) \quad \left[\begin{array}{ll} \min \sum_{j=1}^{n} x_j & \\ s.t. & Ax \leq b \\ & \underline{v} \leq cx^* \leq \overline{v} \\ & x_j \in [0,1] \quad j \in N \end{array} \right. \qquad (\overline{P}) \quad \left[\begin{array}{ll} \max \sum_{j=1}^{n} x_j & \\ s.t. & Ax \leq b \\ & \underline{v} \leq cx^* \leq \overline{v} \\ & x_j \in [0,1] \quad j \in N \end{array} \right.$$

avec x^* qui représente la meilleure solution réalisable courante.

Les contraintes (5.8) et (5.9) sont adaptées aux problèmes réduits. Leur utilisation permet dans certains cas d'accélérer la résolution des problèmes réduits en stoppant l'exploration de l'arbre de recherche plus tôt. Notons également que les bornes $\underline{\sigma}$, $\overline{\sigma}$ sont mises à jour à chaque fois qu'un changement se produit. Une amélioration de $\underline{v}$ entraîne par exemple un recalcul de $\underline{\sigma}$ et $\overline{\sigma}$.

Pour conclure cette section notons que nous utilisons aussi la propriété de fixation classique mentionnée dans le chapitre 3 (section 3.1.3). Cette propriété permet dans certains cas de fixer un ensemble de variables à leur valeur optimale, et ainsi accélérer l'algorithme dans les itérations suivantes. Nous essayons de fixer des variables à chaque fois que nous améliorons la valeur de la borne inférieure. Nous proposons une autre version de l'algorithme dans laquelle nous n'ajoutons pas toutes les contraintes générées au cours de l'exécution de l'algorithme. Nous conservons de cette manière un contrôle sur la taille des problèmes réduits grâce à l'utilisation d'un principe similaire aux contraintes surrogates.

5.2.3 Contrôle de la taille des problèmes réduits

Nous proposons dans cette section un mécanisme simple permettant de contrôler la taille des problèmes réduits à résoudre lors de l'exécution de *HIPL*. Soit p un paramètre dont la valeur correspond au nombre maximum de variables souhaitées dans un problème réduit. Les contraintes sont ajoutées dans le problème tant que leur nombre n'atteint pas p. Une fois qu'il est atteint, les contraintes les plus anciennes sont remplacées par une contrainte de type surrogate (Glover [Glo68]). Plus formellement, à une itération $k > p$, le problème P^k contient les contraintes suivantes

$$f^l x \le |J^1(\overline{x}^l)| \qquad \text{pour } l = k\text{-}p\text{+}2, \ldots, k \tag{5.10-a}$$

$$\sum_{l=1}^{k-p+1} \mu_l f^l x \le \sum_{l=1}^{k-p+1} \mu_l \left| J^1\left(\overline{x}^l\right) \right| - (k - p + 1) \tag{5.10-b}$$

Les contraintes (5.10-a) sont générées entre les itérations k-p+2 et k, et la contrainte surrogate (5.10-b) est créée comme une combinaison linéaire des contraintes générées entre les itérations 1 et k-p+1. Notons que lorsque $p = 1$ une seule contrainte de type (5.10-b) est dans le problème. Lorsque $k \le p$ toutes les contraintes sont ajoutées dans le problème. Dans nos expérimentations le vecteur μ de la contrainte (5.10-b) est égal au vecteur unité.

5.3 Nouvelles heuristiques basées sur la relaxation en nombres entiers mixtes

Comme nous l'avons montré précédemment, la convergence de *HIPL* est généralement lente en pratique. Cela est en particulier le cas lorsque le saut entre la relaxation en continu du problème initial et sa valeur optimale est important. Nous proposons dans cette section des nouvelles heuristiques pour réduire l'écart entre les deux bornes finales obtenues. Cela est rendu possible par l'introduction d'une relaxation en nombres entiers mixtes pour améliorer la qualité de la borne supérieure. Nous améliorons aussi la qualité de la borne inférieure dans la majorité des cas.

5.3.1 Heuristique itérative basée sur la relaxation en nombres entiers mixtes

Dans la suite nous considérerons la relaxation en nombres entiers mixtes (*MIP*) relative au problème *P* et à un sous ensemble *J* de *N,* définie comme suit

$$MIP(P,J)\quad\begin{cases}\max & cx\\ s.c. & Ax\le b\\ & x_j\in\{0,1\} \quad \forall j\in J\\ & x_j\in[0,1] \quad \forall j\in N-J\end{cases}$$

Un sous-ensemble de variables est « forcé » à être binaire dans le problème initial. La taille de ce sous-ensemble est petite par rapport à *n*, et les autres variables sont continues. La proposition suivante justifie l'intérêt de cette relaxation.

Proposition 5.7 :

Pour tous sous-ensembles *J*, *J'* de *N* vérifiant $J' \subseteq J \subseteq N$, on a

$$v(P) \le v(MIP(P,J)) \le v(MIP(P,J')) \le v(LP(P)) \qquad (5.11)$$

Preuve :

Comme $J' \subseteq J$, le problème *MIP*(*P*,*J'*) est une relaxation du problème *MIP*(*P*,*J*), donc nous avons $v(MIP(P,J)) \le v(MIP(P,J'))$. Les autres inégalités se déduisent facilement en observant que $v(MIP(P,N)) = v(P)$ et $v(MIP(P,\varnothing)) = v(LP(P))$. □

En pratique, la relaxation en nombres entiers mixtes fournit généralement de meilleures bornes que la relaxation en continu du problème. Nous proposons plusieurs intégrations de cette relaxation dans l'algorithme *HIPL*. Une première heuristique est obtenue à partir de *HIPL* en remplaçant la relaxation en continu par la relaxation en nombres entiers mixtes. L'algorithme 5.5 décrit l'approche obtenue. Nous l'appelons heuristique itérative basée sur la relaxation en nombres entiers mixtes (*HIMIP*).

Algorithme 5.5 Heuristique itérative basée sur la relaxation en nombre entiers mixtes

Entrée : le problème P.
Sortie : la solution optimale du problème x^*.

1 : Soit x^* une solution réalisable de P;
2 : $v^* = cx^*$; $Q = P$;
3 : Soit $\bar{x}$ une solution optimale de $LP(Q)$; $\bar{v} = v(LP(Q)) = c\bar{x}$;
4 : $\tilde{x} = \bar{x}$;
5 : **tant que** $\lfloor \bar{v} - v^* \rfloor \geq 1$ **faire**
6 : Soit $\tilde{x}$ une solution optimale de $MIP(Q, J^*(\tilde{x}))$;
7 : **si** $(c\tilde{x} < \bar{v})$ **alors** $\bar{v} = c\tilde{x}$;
8 : Soit x une solution optimale de $P(\tilde{x}, J(\tilde{x}))$;
9 : **si** $(cx > v^*)$ **alors** $x^* = x$; $v^* = cx$;
10 : $Q = (Q \mid \{ fx \leq |J^1(\tilde{x})| - 1\})$;
11 : **fin tantque**

Dans cet algorithme et dans la suite de ce chapitre, nous notons $\bar{x}$ (resp. $\bar{x}^k$) une solution optimale de la relaxation en continu (resp. à l'itération k), et $\tilde{x}$ (resp. $\tilde{x}^k$) une solution optimale de la relaxation en nombres entiers mixtes du problème (resp. à l'itération k).

Nous pouvons décomposer l'algorithme *HIMIP* en deux phases. Dans la première phase (lignes 3 et 4) nous résolvons la relaxation en continu du problème de manière à en récupérer une solution optimale, ainsi qu'une borne supérieure. Cette première phase permet de générer la première relaxation en nombres entiers mixtes. Nous choisissons alors comme variables binaires les variables fractionnaires de la solution $\bar{x}$. Le problème réduit obtenu est ainsi clairement disjoint de celui généré lors de la première itération de *HIPL*. Dans la suite de l'algorithme, nous n'utilisons plus que la relaxation en nombres entiers mixtes. Celle-ci change à chaque itération et est définie en fonction de l'ensemble des variables fractionnaires de la solution optimale $\tilde{x}$ de l'itération précédente. La solution $\tilde{x}$ permet également de générer une solution réalisable x du problème initial en résolvant de manière optimale le problème réduit $P(\tilde{x}, J(\tilde{x}))$.

Une remarque importante sur cet algorithme est que les bornes supérieures obtenues ne forment plus nécessairement une suite non-croissante. En effet, la borne supérieure obtenue à une itération donnée dépend de l'ensemble des variables fractionnaires de l'itération précédente et des contraintes ajoutées depuis le début du processus. En fonction du nombre de variables binaires dans la relaxation, la qualité de la borne supérieure peut être moins bonne d'une itération sur l'autre. Nous illustrons ce point dans la section 5.4. Une autre remarque est que les problèmes réduits obtenus entre deux itérations consécutives par *HIMIP* sont totalement disjoints

$$J*\left(\tilde{x}^{k}\right) \cap J*\left(\tilde{x}^{k+1}\right) = \varnothing \tag{5.12}$$

HIMIP permet ainsi de diversifier la recherche. Cet aspect n'est pas présent dans l'heuristique *HIPL* car il n'est pas possible de tirer de conclusions sur les ensembles fractionnaires des solutions de la relaxation en continu au cours de la recherche.

Ajoutons que les améliorations apportées à *HIPL* présentées dans la section 5.2 peuvent être étendues à toutes les heuristiques présentées ici (propriété de dominance, de fixation, *etc*) et que la convergence de *HIMIP* reste valide. Nous proposons de coupler les deux relaxations de manière à obtenir un processus plus robuste et plus performant.

5.3.2 Heuristiques itératives basées sur des relaxations

L'utilisation conjointe des deux relaxations permet d'assurer la non-croissance de la séquence de bornes supérieures au cours du processus, comme dans *HIPL*. Cela permet aussi de générer plus de contraintes pour le problème, et plus de solutions réalisables. La meilleure borne inférieure est dans la majorité des cas améliorée (lorsque le nombre d'itérations est le même pour tous les algorithmes). Les heuristiques itératives basées sur des relaxations (*HIR*) respectent le schéma général suivant.

1) A chaque itération du processus, résoudre une ou plusieurs relaxation(s) du problème courant de manière à générer une ou plusieurs contrainte(s).
2) Résoudre un ou plusieurs problème(s) réduit(s) induit(s) par la (les) solution(s) optimale(s) de la (des) relaxation(s) pour obtenir une ou plusieurs solution(s) réalisable(s).
3) Si le critère d'arrêt choisi est satisfait retourner la meilleure borne inférieure. Sinon ajouter la ou les contraintes générées au problème *P* et retourner à l'étape 1.

La figure 5.1 illustre une instance des heuristiques basées sur des relaxations. Dans cet algorithme nous résolvons deux relaxations et deux problèmes réduits à chaque itération. Nous obtenons deux solutions réalisables et deux contraintes. A l'itération *k* la relaxation en nombres entiers mixtes est obtenue en fonction de la solution de la relaxation en continu $\bar{x}^k$ pour assurer que la borne supérieure correspondante soit au moins aussi bonne que la précédente (proposition 5.7). A l'itération suivante les deux contraintes générées et représentées dans la figure par les vecteurs $\bar{f}$ et $\tilde{f}$ sont ajoutées dans le problème.

Nous proposons un dernier groupe d'heuristiques dans lequel les deux relaxations sont utilisées différemment et indépendamment. Nous utilisons une phase initiale pour définir la première relaxation en nombres entiers mixtes. Les deux relaxations sont ensuite appliquées de manière simultanée (l'une après l'autre d'un point de vue algorithmique). La figure 5.2 présente une version de ces heuristiques que nous appelons heuristiques itératives basées sur des relaxations indépendantes (*HIRI*). Dans la version de la figure 5.2 chacune des deux contraintes générées est ajoutée dans le problème correspondant.

Ajoutons que nous ne mentionnons pas dans les figures 5.1 et 5.2 les cas correspondants à une solution entière pour une des relaxations ou d'un problème non réalisable. Si une des ces conditions est rencontrée le processus de résolution est stoppé d'après le théorème 5.2. De même l'initialisation des variables *k*, *v** n'est pas non plus mentionnée pour ne pas surcharger les figures.

Dans une autre version nous ne considérons qu'un seul problème réduit par itération. Il est obtenu à partir de l'information générée par les deux solutions $\bar{x}^k$ et $\tilde{x}^k$ comme suit

$$P(x, J) \text{ avec } x = \bar{x}^k \text{ ou } x = \tilde{x}^k \text{ et } J = J(\tilde{x}^k) \cup J(\tilde{x}^k) \qquad (5.13)$$

Notons que la taille de *J* peut être contrôlée à l'aide d'un paramètre. La complexité de ces heuristiques est clairement supérieure à celle de *HIPL* puisque nous résolvons plus de relaxations et plus de problèmes réduits. Cependant nous espérons obtenir de meilleures bornes supérieures et inférieures en un nombre d'itérations moins important et donc compenser cette augmentation. Les résultats numériques de la section 5.5 montrent l'impact positif de ces heuristiques.

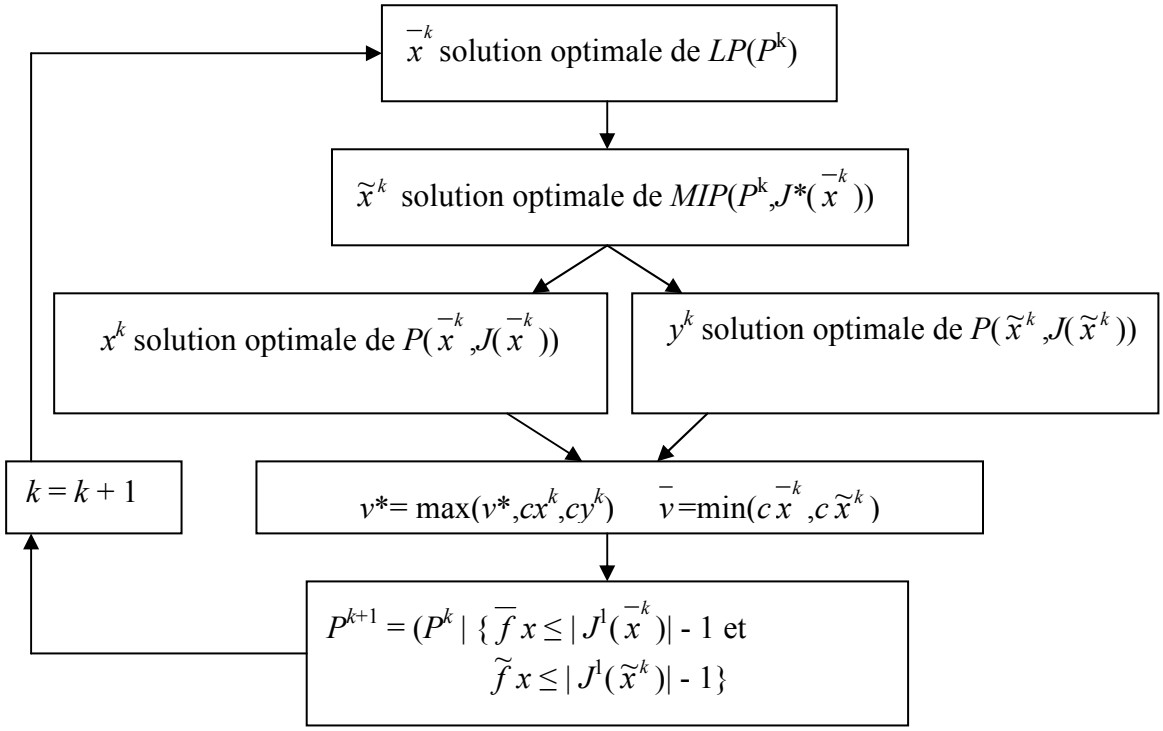

Figure 5.1 – Une instance des heuristiques basées sur des relaxations (*HIR*).

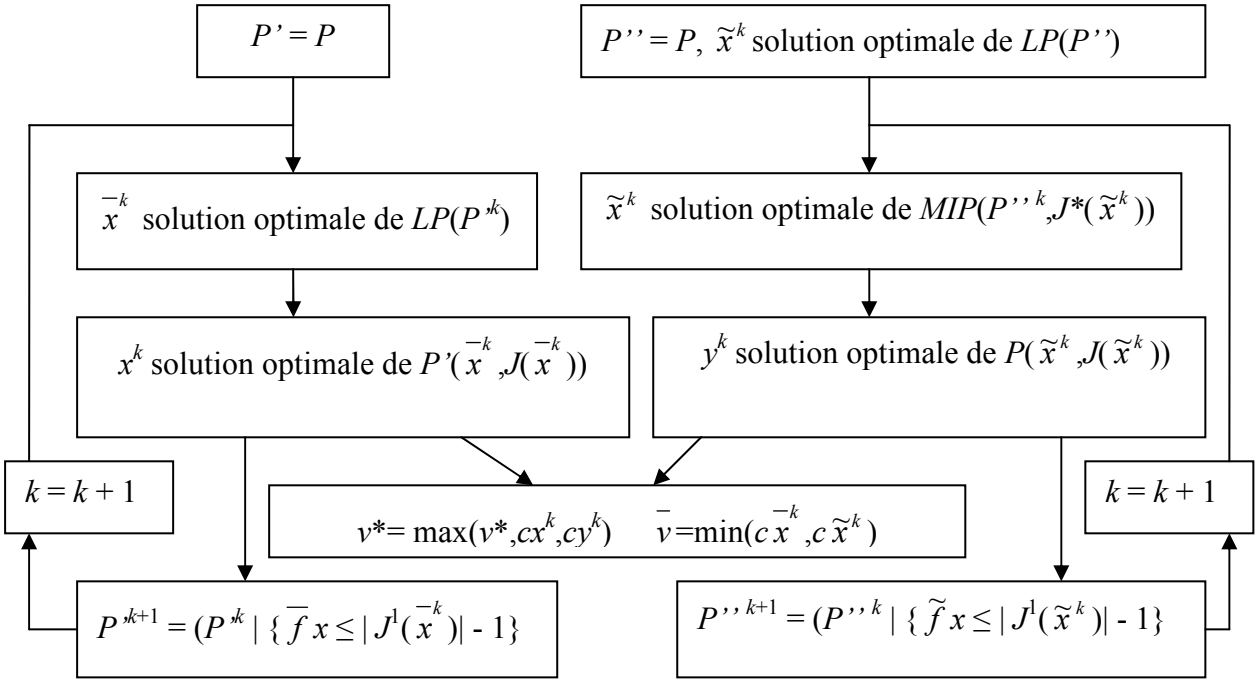

Figure 5.2 – Une instance des heuristiques basées sur des relaxations indépendantes (*HIRI*).

5.4 Illustration

Nous présentons dans cette section les résultats obtenus par les différents algorithmes décrits dans les sections précédentes sur deux instances de tailles moyennes de problème de sac-à-dos multidimensionnel. La première instance, *GK9*, est due à Glover et Kochenberger [GK96]. Elle comporte 30 variables et 10 contraintes. La seconde, *OR-100-5.4*, comporte 100 variables et 5 contraintes et est disponible sur Internet [Bea90]. Nous présentons dans cette section les résultats obtenus par nos algorithmes les plus performants. Nous utilisons dans la suite de ce chapitre les notations suivantes pour décrire les algorithmes.

- *HIPL* : l'heuristique itérative basée sur la programmation linéaire.
- *HIPL_D* : *HIPL* en intégrant la proposition 5.5 concernant la dominance.
- *HIPL_D**(*t*) : *HIPL* en intégrant les propositions 5.5 et 5.6 de dominance. Le paramètre *t* représente la taille maximale d'un problème réduit dans la proposition 5.6.
- *HIMIP* : l'heuristique itérative basée sur la relaxation en nombres entiers mixtes.
- *HIR* : l'heuristique itérative basée sur des relaxations. Les contraintes générées par les deux relaxations sont ajoutées au problème à chaque itération.
- *HIRI* : l'heuristique itérative basée sur des relaxations indépendantes. Les contraintes générées par les deux relaxations sont ajoutées dans les deux problèmes.

5.4.1 Contribution de la dominance

Nous présentons dans le tableau 5.4 les résultats obtenus par l'algorithme *HIPL* (colonne *HIPL*) et par les algorithmes utilisant la dominance (colonnes *HIPL_D*, *HIPL_D**(12) et *HIPL_D**(15)) pour l'instance *GK9*. Les algorithmes utilisant les propriétés de dominance sont lancés avec un maximum de 15 itérations. Pour chaque algorithme nous précisons la taille du problème réduit résolu (colonne $|J^*(\bar{x})|$) et la valeur de la borne inférieure obtenue (colonne *cx*) pour chaque itération (colonne *iter*).

Nous pouvons voir dans le tableau 5.4 que l'ajout de la première propriété de dominance permet de diminuer nettement le nombre de résolutions exactes (de 13 à 6) pour pouvoir obtenir une solution optimale du problème. Lorsque nous utilisons les deux propriétés de dominance, il faut ajuster la valeur du paramètre *t*. Nous avons testé deux valeurs ici, 12 et 15. Nous pouvons noter que le gain en nombre de résolutions exactes est important. Au niveau du temps d'exécution il est clair que sur cette instance il n'y a ni réel gain ni réelle perte car le processus est très rapide. Notons pour terminer qu'un problème réduit de petite

taille n'est pas obligatoirement dominé comme celui correspondant à l'itération 5 dans la colonne *HIPL_D* qui comporte 7 variables.

HIPL			*HIPL_D*			*HIPL_D**(12)			*HIPL_D**(15)		
iter	$\lvert J^*(\bar{x})\rvert$	*cx*	*iter*	$\lvert J^*(\bar{x})\rvert$	*cx*	*iter*	$\lvert J^*(\bar{x})\rvert$	*cx*	*iter*	$\lvert J^*(\bar{x})\rvert$	*cx*
1	6	336	1	9	368	1	11	368	1	15	372
2	7	368	2	10	368	2	11	368	2	14	376
3	8	360	3	11	368	3	13	372			
4	9	368	4	13	372	4	11	376			
5	10	368	5	7	364						
6	6	352	6	9	376						
7	9	368									
8	10	368									
9	10	372									
10	11	368									
11	13	372									
12	7	364									
13	9	376									

Tableau 5.4 – Application de la dominance sur *GK*9.

Le tableau 5.5 présente les résultats pour l'instance *OR-100-5.4*. L'exécution est limitée à 30 itérations pour les versions avec la dominance. Nous donnons les résultats obtenus par *HIPL* lors des 20 premières itérations.

HIPL			*HIPL_D*			*HIPL_D**(15)			*HIPL_D**(20)		
iter	$\lvert J^*(\bar{x})\rvert$	*cx*	*iter*	$\lvert J^*(\bar{x})\rvert$	*cx*	*iter*	$\lvert J^*(\bar{x})\rvert$	*cx*	*iter*	$\lvert J^*(\bar{x})\rvert$	*cx*
1	5	22554	1	8	23303	1	15	23379	1	20	23534
2	5	22983	2	10	23236	2	15	23477	2	19	23473
3	7	23056	3	10	23247	3	15	23379	3	18	**23534**
4	6	22606	4	11	23206	4	12	23379			
5	7	23206	5	10	23344	5	13	23391			
6	8	23001	6	11	23477	6	13	23356			
7	9	23044	7	12	23281	7	14	23388			
8	8	23236	8	13	23318	8	15	23390			
9	8	23303	9	11	23379	9	14	23395			
10	10	23236	10	11	23247	10	14	23300			
11	10	23247	11	12	23379	11	14	23424			
12	11	23206	12	12	23379	12	14	23477			
13	10	23379	13	13	23379	13	15	23477			
14	10	23344	14	13	23391						
15	11	23477	15	13	23356						
16	12	23281	16	14	23388						
17	13	23318	17	15	23390						
18	11	23379									
19	11	23247									
20	12	23379									

Tableau 5.5 – Application de la dominance sur *OR-100-5.4*.

Les résultats du tableau 5.4 sont confirmés pour cette seconde instance. Nous effectuons 17 résolutions exactes au lieu des 30 avec *HIPL_D*. Lorsque nous lançons

*HIPL_D**(15) nous effectuons seulement 13 itérations. Finalement notons le très petit nombre de résolutions exactes effectuées pour *HIPL_D**(20). Il est clair que cette version peut devenir plus coûteuse que la version originale si la valeur du paramètre est trop importante en raison de la taille des problèmes réduits qui augmente. Remarquons aussi que cet algorithme obtient une solution optimale (en gras dans le tableau), ce qui n'est pas le cas pour les autres versions présentées dans ce tableau (même si elle ne peut être prouvée optimale). Ces deux exemples permettent de vérifier qu'il existe en pratique des problèmes réduits dominés résolus inutilement au cours de l'application de *HIPL*.

5.4.2 Relaxation en continu versus relaxation en nombre entiers mixtes

Nous présentons dans le tableau 5.6 les résultats obtenus par les algorithmes *HIPL*, *HIMIP*, *HIR* et *HIRI* sur l'instance *GK*9. Pour chacune des ces heuristiques nous donnons la valeur de la borne supérieure et la valeur de la solution réalisable obtenue à chaque itération. Nous précisons aussi la taille du problème réduit dans le cas de *HIMIP* et de *HIPL*.

iter	*HIPL*			*HIMIP*			*HIRI*		*HIR*	
	$\underline{v}$	*cx*	$\lvert J^*(\bar{x})\rvert$	$\underline{v}$	*cx*	$\lvert J^*(\bar{x})\rvert$	$\underline{v}$	*cx*	$\underline{v}$	*cx*
1	380.3	336	6	380.3	336	6	380	368	379.5	360
2	379.7	368	7	379.8	336	4	379.56	368	378.5	368
3	379.6	360	8	378.2	372	4	377.92	368	377.6	360
4	379.5	368	9	*378.0*	364	3	378.1	364	**376.7**	368
5	378.7	368	10	*378.7*	368	4	377	368	376.6	368
6	378.3	352	6	377.2	**376**	5	377.7	364	376.4	**376**
7	378.2	368	9	**376.2**	356	3	**376.4**	368		
8	377.9	368	10				376.6	368		
9	377.6	372	10				376.5	**376**		
10	377.2	368	11							
11	**376.9**	372	13							
12	376.5	364	7							
13	376.4	**376**	9							

Tableau 5.6 – Application de la relaxation en nombres entiers mixtes sur *GK9*.

HIMIP permet d'obtenir une solution optimale de valeur 376 en 6 itérations. Elle est prouvée comme étant optimale à l'itération suivante car la borne supérieure peut alors être arrondie à 376. Cela représente un gain de l'ordre de 50% par rapport à *HIPL* en nombre d'itérations. Notons que, comme nous l'avons souligné dans la section 5.3.1, les bornes supérieures obtenues ne sont pas nécessairement décroissantes. Ainsi par exemple entre les itérations 4 et 5 (valeurs en italique), la valeur de la borne supérieure augmente (même si cette augmentation n'a pas d'influence ici puisque nous gardons la partie entière inférieure). Notons aussi que sur cet exemple la taille des problèmes réduits résolus par *HIMIP* a tendance

à être inférieure à celle des problèmes réduits résolus par *HIPL*. L'heuristique *HIRI* permet également de gagner quelques itérations par rapport à *HIPL* puisqu'une solution optimale est obtenue à l'itération 9. Elle est prouvée comme étant optimale dès cette itération car la borne supérieure est égale à 376. Notons que le fait de ne pas obtenir une solution de valeur 376 à l'itération 6 comme dans *HIMIP* s'explique ici par le fait que nous ajoutons les contraintes générées par les deux relaxations dans le problème. Les problèmes réduits générés à chaque itération peuvent ainsi être différents d'une heuristique à une autre. Il en est de même pour les valeurs des bornes supérieures. Finalement sur cette instance la version la plus efficace est l'heuristique *HIR*. Nous pouvons en effet obtenir une solution optimale en 6 itérations et prouver que cette solution est optimale à cette même itération grâce à l'amélioration de la borne supérieure. Notons que les bornes supérieures obtenues à la première itération par les heuristiques *HIR* et *HIRI* sont différentes. Cela s'explique par le fait que dans la version de *HIRI* présentée ici nous ajoutons la contrainte associée à la solution $\bar{x}$ initiale dans le problème avant de résoudre la relaxation en nombres entiers mixtes. Cela n'est pas le cas pour l'heuristique *HIR*. Les deux relaxations en nombres entiers mixtes sont donc différentes.

Ces résultats montrent l'influence positive de nos propositions sur ces instances de taille moyenne. Nous abordons dans la section suivante les résultats numériques sur les plus grandes instances.

5.5 Résultats numériques

Les heuristiques itératives basées sur des relaxations proposées dans ce chapitre ont été testées sur deux ensembles d'instances de problème de sac-à-dos multidimensionnel : les instances de la OR-Librairie [Bea90] et les instances générées par Glover et Kochenberger [GK96]. Tous les temps d'exécution référencés sont exprimés en secondes. Les résultats correspondent aux algorithmes les plus efficaces et les valeurs sont comparées à celles générées par CPLEX9.0, celles rapportées par Vasquez et Hao [VH01b] ou Vasquez et Vimont [VV05]. La comparaison avec CPLEX correspond à une exécution de CPLEX avec un temps total du même ordre que celui de nos algorithmes.

5.5.1 Améliorations de *HIPL*

Nous commençons par comparer les résultats de l'algorithme *HIPL* avec le logiciel CPLEX sur le premier ensemble de 270 instances. Le tableau 5.7 présente les résultats obtenus avec un nombre maximum d'itérations (*Max_Iter*) fixé à 50 et 100 pour *HIPL*. Nous

donnons dans ce tableau pour chaque classe d'instances le saut moyen entre la solution obtenue par *HIPL* et celle de CPLEX, i.e. ((*valeur_cplex – valeur_hipl*)*/valeur_cplex*)*100 (%*cx*), et le temps d'exécution (*CPU*).

		Max_Iter = 50		*Max_iter* = 100	
n	*m*	*%cx*	*CPU*	*%cx*	*CPU*
100	5	0.020	1.3	0	9.5
	10	0.018	6.7	0	43.6
	30	0.004	389.7	0.001	1586.1
250	5	0.032	1.7	0.006	31.1
	10	0.038	10.1	0.014	147.9
	30	0.028	1384.5	0.011	4283.7
500	5	0.020	1.9	0.005	65.3
	10	0.046	11.8	0.016	274.7
	30	0.040	2072.4	0.022	5107.6

Tableau 5.7 – Résultats obtenus par *HIPL* par rapport à CPLEX pour la OR-Librairie.

Le tableau 5.7 montre que la qualité des solutions augmente logiquement avec le nombre d'itérations, tout comme le temps d'exécution. Celui-ci devient nettement plus important pour $m = 30$. Les instances avec $n = 100$ sont pratiquement toutes résolues par *HIPL* lorsque *Max_Iter* = 100, et la qualité de la solution finale est bonne par rapport à CPLEX pour tous les autres types d'instances.

Nous présentons dans le tableau 5.8 les résultats obtenus par l'algorithme *HIPL_D* utilisant la proposition 5.5 de dominance. Nous donnons les résultats par classe d'instances (*n* et *m*). La colonne %*R* (resp. %*CPU*) précise le gain en pourcentage de problèmes réduits résolus exactement (resp. de temps d'exécution) pour trois valeurs distinctes de *Max_Iter* : 25, 50 et 100. La comparaison est effectuée avec l'algorithme *HIPL* exécuté seul.

Max_Iter =		25		50		100	
n	*m*	*%R*	*%CPU*	*%R*	*%CPU*	*%R*	*%CPU*
100	5	33.20	19.68	32.40	-3.62	29.93	-27.63
	10	22.93	4.00	22.07	1.98	21.40	-1.07
	30	10.53	9.95	14.20	8.71	15.33	1.66
250	5	32.93	16.79	32.20	-2.11	29.53	0.78
	10	19.33	7.63	21.60	10.99	22.03	15.47
	30	5.73	28.68	9.53	21.78	11.93	10.30
500	5	32.80	10.81	32.40	0.25	30.63	-9.86
	10	20.27	6.31	24.00	11.23	21.53	17.28
	30	4.53	21.13	7.20	13.73	9.57	7.32

Tableau 5.8 – Résultats obtenus par *HIPL_D* sur les instances de la OR-Librairie.

Le tableau 5.8 montre que le gain en nombre de résolutions exactes n'est pas négligeable, même s'il a tendance à diminuer avec le nombre d'itérations et avec *m*. Notons toutefois que pour *m* = 30 le gain augmente avec les itérations. Il ne diminue par contre pas nécessairement avec *n*. Au niveau du gain en temps d'exécution, nous pouvons voir qu'il varie assez fortement, allant d'environ -28% à +29%. Commençons par justifier la présence de nombres négatifs. Ils sont présents essentiellement pour *m* = 5 lorsque le nombre d'itérations augmente. La perte de temps se justifie de la manière suivante. Les résolutions exactes associées à ces instances sont généralement effectuées sur des problèmes réduits de petite taille. Le tableau 5.1 montre par exemple que pour une instance de taille *n* = 100 et *m* = 5, les problèmes réduits n'excèdent pas 25 variables pour 100 itérations. Cela implique que l'ensemble des résolutions peut être fait rapidement par le logiciel CPLEX. L'apport de la dominance ne compense pas le coût supplémentaire lié à la détection des éléments dominés. Cependant en se référant au tableau 5.7 nous pouvons constater que le temps moyen pour les instances avec *n* = 100 et *m* = 5 et 100 itérations est de l'ordre de 10 secondes, et celui associé aux instances ayant 500 variables et 5 contraintes est de l'ordre de 65 secondes. Ainsi les gains (ou pertes) de temps d'exécution sont aussi à relativiser en fonction de la taille des instances. Les valeurs obtenues pour *n* = 250 et 500 et *m* = 10 et 30 sont de ce fait intéressantes car nous obtenons un gain moyen en temps d'exécution de l'ordre de 13% pour 100 itérations, et la moyenne générale est de 7.5%.

Contrairement à l'algorithme *HIPL_D*, l'heuristique *HIPL_D** utilisant les deux propriétés de dominance peut générer de meilleures solutions que *HIPL* (voir tableau 5.5 par exemple). Nous donnons dans le tableau 5.9 les résultats obtenus par cet algorithme pour les instances avec *m* = 30. Les résultats obtenus pour des valeurs de *m* = 5 ou 10 correspondent généralement à une hausse du temps d'exécution par rapport à l'algorithme *HIPL*. Pour ces instances comportant 5 ou 10 contraintes, l'augmentation du temps d'exécution de l'algorithme s'amplifie fortement lorsque *t* augmente. Pour obtenir un gain ou une faible perte en temps d'exécution il faut utiliser une petite valeur de *t* et effectuer peu d'itérations. Cela correspond alors à un gain en nombre de problèmes réduits à résoudre de l'ordre de 30%. L'amélioration de la qualité de la solution finale se produit essentiellement pour les instances avec *m* = 5 ou 10 et elle a tendance à augmenter avec *t* et *Max_Iter*. Elle est cependant très faible car elle concerne généralement une petite partie des 30 instances d'une classe donnée.

Nous précisons dans la ligne *Max_Iter* du tableau 5.9 le nombre d'itérations effectuées, et dans la ligne *t* la valeur du paramètre utilisé pour la proposition 5.6 de dominance. Pour chaque valeur de ces deux paramètres, nous donnons les gains obtenus en terme de nombre de résolutions effectuées (colonne %*R*) et en temps d'exécution (*%CPU*). Ces résultats sont des pourcentages moyens obtenus sur les 30 instances de chaque classe.

Max_Iter=	25				50				100			
t =	40		45		45		50		50		60	
n	*%R*	*%CPU*	*%R*	*%CPU*	*%R*	*%CPU*	*%R*	*%CPU*	*%R*	*%CPU*	*%R*	*%CPU*
100	73.46	-21.64	88.08	7.18	71.63	-21.29	88.43	20.99	61.12	-4.83	89.83	51.43
250	51.03	-15.93	76.03	-1.04	50.33	14.01	74.05	46.47	42.77	22.17	78.28	70.43
500	32.56	13.35	61.41	27.58	36.08	18.26	58.89	44.45	31.06	19.76	65.41	60.19

Tableau 5.9 – Résultats obtenus par *HIPL_D** pour les instances avec $m = 30$.

Le tableau 5.9 montre que le gain en temps d'exécution augmente avec le nombre d'itérations pour les instances les plus contraintes. L'augmentation dépend cependant également de la valeur du paramètre *t*. Nous pouvons remarquer que pour deux valeurs consécutives de *Max_Iter* et une valeur de *t* identique les gains donnés par %*R* et *%CPU* diminuent. Ils deviennent réellement intéressants à partir des valeurs *Max_Iter* = 50 et $t = 50$. Contrairement aux instances comportant moins de contraintes nous parvenons ici à obtenir un gain en temps d'exécution important lorsque nous résolvons un petit nombre de problèmes réduits. L'explication logique est qu'ici le coût supplémentaire correspondant à la mise en œuvre de la dominance est compensé par le fait que les problèmes réduits non résolus sont nettement plus difficiles que pour les instances comportant moins de contraintes. Lorsque *Max_Iter* = 100, nous pouvons ainsi obtenir un gain de l'ordre de 60% en temps d'exécution sur l'ensemble des 90 instances par rapport à l'algorithme *HIPL*.

Ces expériences numériques montrent que l'algorithme *HIPL* peut être réellement amélioré grâce aux deux propriétés de dominance. L'efficacité de la première propriété est dans l'ensemble plus importante lorsque le nombre d'itérations est petit. L'influence du paramètre *t* avec la seconde propriété est claire. Pour le problème *SADM*, il semble difficile de donner une valeur de *t* qui assure une amélioration des performances de *HIPL*. Cette valeur doit augmenter avec *Max_Iter* de manière à pouvoir obtenir à la fois un gain en nombre de résolutions exactes et en temps d'exécution. Elle ne doit cependant pas devenir trop importante non plus au risque d'entraîner une baisse des performances de *HIPL_D**.

Pour mesurer l'impact de l'agrégation des contraintes, nous avons testé l'algorithme avec plusieurs valeurs possibles pour le nombre de contraintes (p) à ajouter dans le problème : 1, 5, 10 et 20. Nous présentons les résultats obtenus en terme de qualité de la solution finale par rapport aux valeurs obtenues par CPLEX lorsque *Max_Iter* = 100 dans le tableau 5.10. Chaque colonne est une moyenne sur 30 instances. Nous donnons pour chaque classe d'instances le résultat en termes de qualité de solution (*%cx*) et en temps d'exécution (*CPU*).

	n	100			250			500		
p	m	5	10	30	5	10	30	5	10	30
1	*%*	0.378	0.312	0.1	0.249	0.222	0.094	0.147	0.132	0.09
	T	0.1	0.7	58	0.1	0.8	134	0.1	0.8	258
5	*%*	0.144	0.17	0.047	0.161	0.159	0.077	0.087	0.107	0.069
	T	0.3	1.4	97	0.4	1.6	242	0.4	1.8	455
10	*%*	0.044	0.102	0.011	0.078	0.093	0.048	0.052	0.078	0.057
	T	0.7	3.5	219	0.8	3.7	710	0.9	4.1	1388
20	*%*	0.007	0.02	0.001	0.021	0.039	0.026	0.016	0.044	0.036
	T	3	11.7	659	3.5	15.5	2089	3.5	16	4352

Tableau 5.10 – Résultats obtenus avec l'agrégation de contraintes sur la OR-Librairie.

Nous pouvons voir dans le tableau 5.10 que pour une valeur donnée de p le temps d'exécution augmente logiquement avec n et m. Il en est de même avec la qualité de la meilleure solution (saut moyen de 0.2% pour p = 1 et de 0.02% pour p = 20). Notons que l'augmentation du temps d'exécution entre m = 10 et 30 est très importante, comme pour *HIPL*, quelles que soient les valeurs des autres paramètres. Si nous comparons les résultats obtenus en termes de qualité de solution et de temps d'exécution avec *HIPL* (voir tableau 5.7) nous pouvons surtout remarquer une forte diminution du temps d'exécution, même pour p = 20 et m = 30. La différence de temps d'exécution entre chaque valeur de n est également un peu moins forte. Finalement la qualité de la solution retournée est logiquement un peu moins bonne mais cette approche permet de définir une méthode intermédiaire avec un compromis entre la qualité de la solution et le temps d'exécution intéressant.

5.5.2 Heuristiques itératives basées sur des relaxations

Nous comparons dans cette section les résultats obtenus par les algorithmes *HIPL*, *HIR* et *HIRI*. L'algorithme *HIMIP* s'est révélé être moins performant en terme de qualité de la solution finale lors des expériences numériques préliminaires. Nous ne présentons pas ses résultats pour ne pas surcharger les tableaux. Nous avons testé dans un premier temps les algorithmes sur les instances de la OR-Librairie comportant 100 et 250 variables. Le nombre

d'itérations est fixé à 120 pour m = 5 et 10 et à 60 pour m = 30. Les valeurs obtenues sont comparées à celles de CPLEX. A l'exception de *HIPL* qui est dominée pour les instances de la classe m = 30 et α = 0.5, les trois algorithmes permettent d'obtenir les mêmes solutions que CPLEX sur toutes les instances comportant 100 variables.

Nous présentons dans le tableau 5.11 les résultats obtenus pour n = 250. Pour chaque valeur de m et α nous donnons la valeur moyenne de la solution finale sur les 10 instances de la classe. Lorsqu'un algorithme domine (resp. est dominé par) les trois autres, sa valeur est précisée en gras (resp. en italique). L'algorithme *HIR* domine clairement les trois autres algorithmes sur cet ensemble de 90 instances. La domination entre CPLEX et *HIRI* est plus difficile à établir puisque *HIRI* domine CPLEX deux fois (sur les plus grandes instances), et CPLEX domine *HIRI* trois fois. L'algorithme *HIPL* est clairement dominé mais il est le plus rapide. Les temps d'exécution de *HIR*, qui est l'algorithme le plus lent, sont de l'ordre de 750, 3200 et 3700 secondes respectivement pour m = 5, 10 et 30. Il est clair que les temps d'exécution pour obtenir la meilleure solution sont généralement inférieurs (tout comme pour CPLEX) mais nos algorithmes ne parviennent pas à prouver l'optimalité des solutions sur ces instances (comme CPLEX pour m > 5).

m	α	*CPLEX*	*HIPL*	*HIRI*	*HIR*
5	0.25	60413.5	*60407.8*	60413.5	60413.5
	0.5	109292.8	*109290.3*	109292.8	109292.8
	0.75	151560.3	*151559.3*	151560.2	151560.3
10	0.25	59021.3	*59009.5*	59020.2	59021.3
	0.5	108729.3	*108726.9*	108729.3	108729.3
	0.75	151346.2	*151339.3*	151346.2	151346.2
30	0.25	56936.5	*56902.2*	56929.9	**56937.4**
	0.5	106703.1	*106698.8*	106707.7	**106709.8**
	0.75	150480.2	*150467.3*	150482.5	150482.5

Tableau 5.11 – Résultats obtenus par *HIPL*, *HIR* et *HIRI* pour n = 250.

Dans les trois tableaux 5.12, 5.13 et 5.14 nous donnons les résultats obtenus par nos algorithmes sur les 90 instances les plus difficiles comportant 500 variables. Les algorithmes ont été lancés pour 120, 100 et 60 itérations chacun pour m = 5, 10 et 30 respectivement. Pour chaque instance nous donnons dans la colonne v^* la meilleure valeur connue répertoriée par Vasquez et Vimont [VV05]. Pour chacune des trois heuristiques nous donnons la différence entre la meilleure solution connue et la solution finale ($v^* - \underline{v}$), la différence entre la borne

supérieure finale et la borne inférieure ($\bar{v} - \underline{v}$), et le temps correspondant pour obtenir cette solution (*CPU**).

Problème	v^*	*HIPL*			*HIRI*			*HIR*		
		v^*-$\underline{v}$	$\bar{v} - \underline{v}$	*CPU**	v^*-$\underline{v}$	$\bar{v} - \underline{v}$	*CPU**	v^*-$\underline{v}$	$\bar{v} - \underline{v}$	*CPU**
1	120148	19	83	119	0	59	1303	0	55	2906
2	117879	16	69	54	0	47	56	0	43	1145
3	121131	0	59	73	0	54	79	0	51	107
4	120804	10	71	33	5	61	471	0	49	3671
5	122319	0	77	4	0	71	4	0	67	8
6	122024	13	86	147	0	64	1303	0	61	1363
7	119127	0	68	19	0	62	130	0	62	117
8	120568	0	55	17	0	48	19	0	45	39
9	121575	0	67	11	0	64	14	**-11**	51	3767
10	120717	0	61	146	0	55	651	0	50	113
11	218428	2	53	253	0	47	5382	0	45	5730
12	221202	11	58	1	0	43	29	0	24	44
13	217542	8	63	47	6	52	1141	6	51	2337
14	223560	0	69	10	0	62	121	0	55	26
15	218966	4	74	1	0	62	294	0	50	582
16	220530	3	67	66	3	58	676	3	52	387
17	219989	7	69	7	0	51	308	0	45	315
18	218215	20	68	81	0	39	9	0	32	22
19	216976	0	62	1	0	46	4	0	44	12
20	219719	8	78	27	0	64	206	0	55	257
21	295828	0	45	1	0	34	2	0	28	1
22	308086	0	51	66	0	38	555	0	26	428
23	299796	0	54	9	0	47	2	0	43	2
24	306480	4	62	3	0	38	1103	0	33	318
25	300342	0	48	1	0	39	12	0	34	4
26	302571	9	70	54	0	50	393	0	40	189
27	301339	10	51	3	0	31	525	0	19	343
28	306454	0	41	14	0	34	69	0	16	26
29	302828	0	47	42	0	42	481	0	32	222
30	299910	0	47	78	0	34	1329	0	35	926

Tableau 5.12 – Résultats obtenus sur les instances OR-500-5.

Nous pouvons voir dans le tableau 5.12 que l'heuristique *HIR* permet d'obtenir les meilleures solutions pour 27 instances, et permet d'améliorer une meilleure solution. L'heuristique *HIPL* permet l'obtention de 15 meilleures solutions, alors que *HIRI* en obtient 27. L'écart entre la borne supérieure finale et la borne inférieure finale n'est pas très important pour ces instances, mais la convergence reste toutefois impossible à obtenir avec ce nombre d'itérations. Pour donner un ordre d'idée sur le temps d'exécution, les heuristiques *HIRI* et *HIR* prennent en moyenne 2600 et 3000 secondes par instance alors que *HIPL* nécessite environ 190 secondes. Nous avons donc ici la confirmation que les nouveaux algorithmes que nous proposons nécessitent un temps d'exécution plus important lorsque le nombre d'itérations est fixé. Notons toutefois que le temps d'exécution de ces algorithmes

reste raisonnable si on se réfère à l'article de Vasquez et Vimont [VV05] dans lequel les auteurs précisent que le temps moyen par instance est de l'ordre de 50 heures sur un Pentium IV 2Ghz (la RAM n'est pas précisée).

Nous donnons dans le tableau 5.13 les résultats obtenus pour les instances ayant 10 contraintes.

Problème	v^*	*HIPL*			*HIRI*			*HIR*		
		$v^*-\underline{v}$	$\bar{v}-\underline{v}$	*CPU**	$v^*-\underline{v}$	$\bar{v}-\underline{v}$	*CPU**	$v^*-\underline{v}$	$\bar{v}-\underline{v}$	*CPU**
1	117811	2	186	449	2	176	1133	2	172	3086
2	119232	15	197	119	15	188	1685	15	185	2068
3	119215	4	173	569	4	163	643	0	156	5028
4	118813	0	218	105	0	209	65	**-12**	191	3400
5	116509	0	170	70	**-5**	155	829	**-5**	155	241
6	119504	35	214	64	0	168	1988	14	178	3130
7	119827	39	216	378	14	179	3719	33	199	2810
8	118329	20	207	373	6	183	363	**-4**	170	6146
9	117815	39	200	65	34	183	1582	34	182	2200
10	119231	65	251	318	0	178	993	**-20**	156	4282
11	217377	0	154	784	0	138	1708	0	133	9
12	219077	11	167	693	11	149	377	0	137	1727
13	217806	9	168	330	**-41**	97	33	**-41**	98	428
14	216868	17	172	196	0	132	1307	0	136	1032
15	213859	13	149	1114	7	125	3151	6	122	1707
16	215086	0	153	238	0	138	958	0	133	1317
17	217940	14	159	529	9	131	2258	0	122	4606
18	219984	0	167	96	0	144	22	**-6**	141	5729
19	214375	24	185	203	0	145	663	0	142	329
20	220899	47	206	62	27	166	1370	13	149	3174
21	304387	24	164	161	17	135	2508	0	112	5118
22	302379	43	193	60	0	126	1048	0	120	1462
23	302416	0	143	201	0	113	446	0	116	85
24	300757	10	179	160	**-27**	125	535	**-27**	120	2004
25	304374	17	194	546	0	151	4081	0	150	1561
26	301836	40	136	315	40	113	316	0	67	5314
27	304952	3	167	7	0	135	45	0	134	326
28	296478	22	157	22	6	112	2409	6	115	785
29	301359	28	187	42	2	151	1872	2	133	1015
30	307089	17	154	72	6	117	545	0	108	839

Tableau 5.13 – Résultats obtenus sur les instances OR-500-10.

Nous pouvons remarquer dans le tableau 5.13 que l'heuristique *HIR* permet d'obtenir 14 meilleures solutions et d'améliorer 7 meilleures solutions. L'application de *HIPL* permet d'obtenir 6 meilleures solutions. Finalement l'algorithme *HIRI* obtient 12 meilleures solutions et améliore 3 meilleures solutions. L'écart entre la borne supérieure finale et la borne inférieure finale a tendance à augmenter ici de manière importante et les 120 itérations effectuées sont désormais loin de suffire pour pouvoir prouver l'optimalité des solutions

finales. Le temps d'exécution associé aux heuristiques est ici de l'ordre de 730, 4500 et 4800 secondes par instance pour *HIPL*, *HIRI* et *HIR* respectivement contre 70 heures par instance pour Vasquez et Vimont [VV05].

Le tableau 5.14 présente les résultats obtenus sur les plus grandes instances avec m = 30. Ces résultats sont les plus mitigés dans le sens où les heuristiques *HIRI* et *HIR* ne nous permettent d'améliorer que 3 meilleures solutions connues. *HIR* égale 8 meilleures solutions et en améliore 3, et *HIRI* en égale 5 et en améliore 1. Enfin, *HIPL* égale 2 meilleures solutions.

Problème	v^*	*HIPL*			*HIRI*			*HIR*		
		$v^*-\underline{v}$	$\bar{v}-\underline{v}$	*CPU**	$v^*-\underline{v}$	$\bar{v}-\underline{v}$	*CPU**	$v^*-\underline{v}$	$\bar{v}-\underline{v}$	*CPU**
1	116056	109	651	4366	109	598	3162	108	544	747
2	114810	88	624	1026	88	574	515	30	472	4136
3	116712	51	662	123	22	581	2845	**-29**	468	3499
4	115329	86	682	3427	**-25**	508	2660	**-16**	473	548
5	116525	51	593	2183	51	551	1034	9	473	7765
6	115741	0	612	2277	0	560	1492	0	505	652
7	114181	74	566	1659	72	521	1262	70	485	5176
8	114348	132	613	3303	54	503	3469	4	418	3313
9	115419	0	467	1119	0	422	392	0	369	320
10	117116	93	624	2769	0	475	3445	0	429	2990
11	218104	36	514	2795	36	471	378	36	421	250
12	214648	98	508	1359	32	401	234	3	334	3880
13	215978	75	482	1810	36	415	2085	56	386	3431
14	217910	83	506	13	48	427	790	25	372	3771
15	215689	76	465	1047	25	384	305	49	369	3617
16	215890	53	478	1232	23	423	3133	**-29**	335	627
17	215907	107	549	846	28	415	2378	0	346	3631
18	216542	64	514	3772	0	403	1824	32	396	3271
19	217340	27	503	392	27	448	1047	27	405	3544
20	214739	59	516	3264	38	447	2468	49	427	4346
21	301675	48	399	121	32	345	361	0	290	2586
22	300055	41	423	1919	0	343	2232	0	313	1544
23	305087	7	399	1731	7	355	1819	7	336	1208
24	302032	27	433	4302	24	371	916	24	349	587
25	304462	49	467	724	37	408	3096	37	373	4449
26	297012	53	433	1384	24	364	2055	43	354	1264
27	303364	60	444	2162	42	382	2944	35	339	1505
28	307007	63	443	280	8	344	2462	8	328	1320
29	303199	56	443	2585	21	357	777	0	293	1928
30	300572	36	465	3865	36	427	789	0	343	1204

Tableau 5.14 – Résultats obtenus sur les instances OR-500-30.

Pour ces instances nous n'effectuons que 60 itérations en raison de la complexité temporelle de l'approche qui tend à devenir importante. Le temps moyen d'exécution des heuristiques reste ainsi de l'ordre de 3600, 4600 et 5200 secondes pour *HIPL*, *HIRI* et *HIR*

respectivement. Ces temps d'exécution sont raisonnables en comparaison avec les 100 heures mentionnées par instance pour Vasquez et Vimont [VV05]. La différence entre les deux bornes obtenues est très importante ici, et la convergence est clairement inutilisable sur ces instances, même en augmentant fortement le nombre d'itérations. Notons pour terminer que les heuristiques *HIRI* et surtout *HIR* dominent clairement *HIPL*, et que les solutions finales sont proches des meilleures solutions connues.

Comme nous l'avons dit précédemment, l'algorithme *HIPL* est bien plus rapide que les algorithmes *HIR* et *HIRI* pour un nombre d'itérations fixé sur les instances de grande taille. Nous donnons dans le tableau 5.15 un résumé des résultats obtenus sur les 90 instances avec $n = 500$ si nous fixons le temps d'exécution pour les trois algorithmes. Le temps accordé est de 2000, 3500 et 5000 secondes pour $m = 5$, 10 et 30 respectivement. Comme pour le tableau 5.11, une ligne correspond à une moyenne sur 10 instances. Nous donnons pour chaque algorithme et chaque classe d'instances la valeur moyenne de la solution finale obtenue. Un algorithme qui domine (resp. qui est dominé par) les autres algorithmes a sa valeur en gras (resp. en italique).

m	α	*HIPL*	*HIRI*	*HIR*
5	0.25	120628.7	120628.7	**120629.8**
	0.5	**219512.1**	*219510.9*	219511
	0.75	302363.4	*302363.2*	302363.4
10	0.25	*118618.8*	118621.1	**118621.7**
	0.5	217326.5	*217317.4*	**217329.3**
	0.75	*302597*	302598.3	**302600.6**
30	0.25	*115576.2*	115585.4	**115596.3**
	0.5	*216227.8*	**216244.4**	216238.6
	0.75	302420.6	*302419.8*	**302426.8**

Tableau 5.15 – Résultats de *HIPL*, *HIRI* et *HIR* avec le même temps d'exécution.

Le tableau 5.15 montre clairement que l'heuristique *HIR* améliore les résultats obtenus par *HIPL*. *HIR* n'est jamais dominée par les deux autres heuristiques et elle obtient les meilleures moyennes pour 7 classes d'instances sur 9. L'algorithme *HIRI* est un peu moins performant puisqu'il obtient une fois les meilleures solutions et est dominé par les deux autres algorithmes 4 fois. Les résultats de *HIRI* sont assez proches de ceux de *HIPL* qui obtient 2 fois les meilleurs résultats et est dominée 4 fois également.

Devant ces résultats intéressants, nous avons essayé d'améliorer les solutions obtenues en augmentant le nombre d'itérations. Nous présentons les résultats dans le tableau 5.16. La

première colonne précise l'instance testée. Par exemple, 5.500-9 désigne la 9ème instance de la classe n = 500 et m = 5. Lorsque nous mentionnons *N/A* cela signifie que l'algorithme correspondant n'a pas été exécuté sur cette instance ou n'a pas amélioré le résultat précédent. Nous donnons dans la colonne *CPU* le temps total d'exécution de l'algorithme. Les autres données sont identiques aux tableaux 5.11 à 5.13. Le nombre d'itérations des heuristiques est fixé à 200 pour m = 5 et 10 et à 100 pour m = 30. La valeur mentionnée en gras représente une amélioration de la meilleure valeur connue non encore obtenue précédemment.

Probème	*v**	*HIRI*				*HIR*			
		v-v*	*iter**	*CPU**	*CPU*	*v*-v*	*iter**	*CPU**	*CPU*
5.500-9	121575	-11	121	3193	8010	N/A	N/A	N/A	N/A
5.500-13	217542	0	139	4244	8021	0	181	9781	11035
5.500-16	220530	0	153	1469	3840	0	129	973,5	1233
10.500-2	119232	N/A	N/A	N/A	N/A	**-17**	162	10257	13569
10.500-6	119504	N/A	N/A	N/A	N/A	0	127	7064	9222
10.500-7	119827	N/A	N/A	N/A	N/A	0	121	3864	5980
10.500-9	117815	0	141	4743	5299	N/A	N/A	N/A	N/A
10.500-10	119231	-6	138	3320	4060	N/A	N/A	N/A	N/A
10.500-12	219077	0	135	5514	6440	N/A	N/A	N/A	N/A
10.500-20	220899	13	128	3474	4822	12	144	4532	7701
10.500-30	307089	0	131	2093	3093	N/A	N/A	N/A	N/A
30.500-1	116056	47	80	4969	9530	47	76	12480	21341
30.500-2	114810	30	81	4885	9235	N/A	N/A	N/A	N/A
30.500-3	116712	13	68	4146	6131	N/A	N/A	N/A	N/A
30.500-11	218104	31	63	3724	5999	31	72	8974	15111
30.500-12	214648	22	63	3617	5886	N/A	N/A	N/A	N/A
30.500-13	215978	33	75	4408	5944	0	61	7227	15427
30.500-14	217910	35	79	4436	5724	N/A	N/A	N/A	N/A
30.500-15	215689	0	90	5087	5709	0	74	8726	12939
30.500-17	215907	0	94	5690	6066	N/A	N/A	N/A	N/A
30.500.18	216542	N/A	N/A	N/A	N/A	12	63	7754	16369
30.500-19	217340	7	69	4165	6074	22	95	16735	18413
30.500-27	303364	36	86	5148	6009	0	81	9700	13172
30.500-30	300572	32	68	4036	6000	N/A	N/A	N/A	N/A

Tableau 5.16 – Amélioration des résultats de *HIRI* et de *HIR* sur les instances avec m = 500.

L'heuristique *HIR* améliore une nouvelle meilleure solution pour l'instance 10.500-2, alors que nous étions précédemment à 15 unités (voir tableau 5.13). Le temps d'exécution des heuristiques *HIRI* et *HIR* augmente de manière assez importante. L'heuristique *HIRI* améliore également une meilleure solution connue pour l'instance 10.500-10, mais cette solution est moins bonne que celle obtenue précédemment par *HIR* (voir tableau 5.13).

Le tableau 5.17 suivant résume les résultats obtenus sur les 90 plus grandes instances de la OR-Librairie par nos algorithmes *HIRI* et *HIR*. Nous précisons pour chaque valeur de m le nombre de nouvelles meilleures valeurs obtenues (#Améliore), de meilleures valeurs égalées (#Egal) et de valeurs inférieures (#Moins) respectivement.

m	#Améliore	#Egal	#Moins
5	1	29	0
10	8	16	6
30	3	12	15

Tableau 5.17 – Bilan des algorithmes sur les instances avec $n = 500$.

Nous concluons cette section avec le tableau 5.18 qui présente les résultats obtenus sur les instances de Glover et Kochenberger [GK96]. Dans ce tableau nous donnons dans la colonne *iter** l'itération au cours de laquelle chaque algorithme a obtenu sa meilleure solution. Les valeurs mentionnées dans *v** sont celles répertoriées par Vasquez et Hao [VH01b]. Les algorithmes ont été testés avec une limite de 60 itérations pour les instances *GK18* à *MK_gk05*. Pour les autres instances, la limite est fixée à 100 itérations.

Problème	*n*	*m*	*v**	*HIPL*				*HIRI*				*HIR*			
				$v^*-\underline{v}$	$\bar{v}-\underline{v}$	*iter**	*CPU**	$v^*-\underline{v}$	$\bar{v}-\underline{v}$	*iter**	*CPU**	$v^*-\underline{v}$	$\bar{v}-\underline{v}$	*iter**	*CPU**
GK018	100	25	4528	0	16	58	290	0	12	6	79	0	9	11	681
GK019	100	25	3869	0	15	36	66	0	11	6	71	0	8	2	25
GK020	100	25	5180	0	16	45	240	0	12	25	475	0	9	9	366
GK021	100	25	3200	0	17	19	27	0	13	8	58	0	10	8	245
GK022	100	25	2523	0	18	30	61	0	14	14	90	0	10	7	117
GK023	200	15	9235	2	11	36	6	0	8	40	45	0	7	40	185
GK024	500	25	9070	3	12	3	1	1	10	55	1168	0	8	45	2509
MK_gk01	100	15	3766	0	7	9	1	0	5	9	5	0	4	2	2
MK_gk02	100	25	3958	0	15	40	45	0	11	15	51	0	8	10	100
MK_gk03	150	25	5656	1	14	52	458	0	10	35	1925	0	8	3	33
MK_gk04	150	50	5767	0	25	5	223	0	18	3	282	0	15	3	472
MK_gk05	200	25	7560	0	16	49	459	0	13	25	1261	0	12	8	636
MK_gk06	200	50	7677	2	27	19	1727	**-1**	18	98	23994	**-1**	12	36	10042
MK_gk07	500	25	19220	3	15	51	287	1	10	76	8375	1	10	74	15769
MK_gk08	500	50	18806	3	27	44	3998	1	23	4	976	1	21	2	11183
MK_gk09	1500	25	58087	5	18	45	396	**-4**	7	86	15064	-2	8	60	17732
MK_gk10	1500	50	57295	6	29	22	1999	3	22	27	6598	3	20	5	6355
MK_gk11	2500	100	95237	9	49	25	2261	8	43	37	9463	8	43	5	1922

Tableau 5.18 – Résultats obtenus sur les instances de Glover et Kochenberger.

Les résultats obtenus par nos heuristiques sont très encourageants. Sur les 18 instances nous égalons 12 meilleures solutions et nous en améliorons deux (*MK_gk06* et *Mk_gk09*). Les temps d'exécution des heuristiques *HIR* et *HIRI* restent raisonnables par rapport à la taille des instances. Il est inférieur à 2 heures pour les instances *GK18* à *Mk_gk05*. Pour les plus grandes instances il est au plus de 11 heures dans le cas de *HIR* et au plus de 7 heures pour *HIRI*. Remarquons que *HIR* ou *HIRI* obtiennent de meilleures solutions que *HIPL* pour 9 instances, et la même solution pour les 9 autres (les plus petites).

5.6 Conclusions

Nous avons proposé dans ce chapitre une nouvelle présentation d'un algorithme datant de la fin des années 1970 pour la résolution de problèmes en variables 0-1. Celui-ci génère une séquence de bornes supérieures non-croissantes en résolvant une série de relaxation en continu. Il permet aussi d'obtenir une séquence de bornes inférieures en résolvant une série de problèmes réduits. Nous avons inséré plusieurs propriétés sur les problèmes réduits ainsi qu'une nouvelle preuve de la convergence de l'algorithme. Nous avons ensuite proposé plusieurs améliorations de cet algorithme basées sur la mise en évidence d'un phénomène de dominance. Ce concept de dominance permet en pratique de diminuer fortement le nombre de problèmes réduits à résoudre de manière optimale et dans certains cas une diminution du temps d'exécution. Les résultats obtenus montrent cependant que l'utilisation pratique de ces propriétés n'est pas toujours avantageuse, et qu'elle dépend du réglage d'un paramètre. Nous avons également proposé de nouveaux algorithmes. Ceux-ci combinent des méthodes exactes (lors de la résolution des problèmes réduits), et des méthodes approchées (une itération fournit un encadrement de la valeur optimale du problème). Ils reposent sur l'utilisation de la relaxation en nombres entiers mixtes. La principale motivation dans l'intégration de cette relaxation est qu'elle permet d'affiner la valeur de la borne supérieure obtenue à chaque itération. Elle nous permet également de diversifier la recherche en explorant d'autres problèmes réduits. Même si les résultats numériques montrent que la convergence reste peu utilisable en pratique, ces algorithmes nous permettent une nette amélioration de la qualité des bornes supérieures et inférieures dans la majorité des instances des problèmes de sac-à-dos multidimensionnel que nous avons traitées. Nous parvenons ainsi à améliorer 14 meilleures solutions connues sur un ensemble de 108 instances réputées difficiles.

Conclusions générales et perspectives

De nombreux problèmes réels peuvent être modélisés sous la forme d'un programme linéaire en variables 0-1 mixtes. Nous avons proposé dans ce mémoire plusieurs algorithmes pour la résolution de ces problèmes. Nous les avons validé sur le problème du sac-à-dos multidimensionnel. Nous commençons cette section par présenter un comparatif des résultats obtenus par chacune de nos trois approches. Nous reprenons ensuite chacune des méthodes de résolution proposées et récapitulons ses avantages et ses limites, ainsi que ses extensions possibles à d'autres problèmes. Nous précisons également quelques perspectives de développement et de recherche que nous envisageons.

Comparatif des résultats des méthodes de résolution

Nous présentons dans le tableau 6.1 suivant les résultats moyens obtenus par chacune des trois approches proposées dans ce mémoire (intensification globale (*IG*), recherche dispersée (*RD*) et heuristiques basées sur des relaxations (*HR*)) sur les 270 instances de la OR-Librairie. Pour chacune des trois méthodes nous précisons le saut moyen par rapport à la meilleure valeur (colonne *%Meilleure*). La meilleure valeur est celle donnée par Chu et Beasley [CB98] pour n = 100 et 250 et par Vasquez et Vimont [VV05] pour n = 500). Nous précisons aussi le temps d'exécution moyen en secondes (colonne *CPU*). Chaque ligne dans ce tableau est une moyenne sur 30 instances.

n	*m*	*IG*		*RD*		*HR*	
		%Meilleure	*CPU*	*%Meilleure*	*CPU*	*%Meilleure*	*CPU*
100	5	0,01	23	0,00	24	0,00	202
250		0,01	66	0,01	52	-0,01	1325
500		0,02	140	0,01	219	0,00	2958
100	10	0,08	39	0,00	45	0,00	1040
250		0,06	115	0,01	195	-0,03	5373
500		0,07	418	0,04	871	0,00	4800
100	30	0,22	33	-0,01	81	-0,02	2883
250		0,10	110	-0,01	312	-0,06	4264
500		0,13	340	0,06	1338	0,01	5207

Tableau 6.1 – Comparatif des trois approches sur la OR-Librairie.

Ce tableau montre bien que les heuristiques basées sur des relaxations sont les approches les plus performantes en terme de qualité des solutions obtenues. Ce sont cependant les approches les plus coûteuses en terme de temps d'exécution. Nous pouvons aussi noter que le processus d'intensification globale est en moyenne le moins coûteux mais aussi le moins performant. Finalement la recherche dispersée permet d'avoir un compromis entre le temps d'exécution et la qualité des solutions finales. Cette approche obtient de bonnes valeurs en moyenne même si elle est un peu moins performante pour les instances comportant 500 variables.

Recherche tabou : intensification globale avec la programmation dynamique

Le processus d'intensification globale que nous avons défini repose sur la coopération entre la programmation dynamique et la recherche tabou. L'intégration de principes de réduction dans l'algorithme de programmation dynamique permet dans certains cas de réduire de manière conséquente la taille des instances comme nous l'avons montré dans le chapitre 3. Les résultats présentés dans ce chapitre montrent aussi que notre algorithme peut rivaliser avec un logiciel d'optimisation performant sur des instances de grandes tailles, en particulier lorsque le nombre de contraintes est petit. Une limite de l'approche réside dans sa difficulté à résoudre des instances fortement contraintes.

Nous envisageons plusieurs perspectives pour ce travail. L'algorithme de recherche tabou semble être le maillon le plus facilement améliorable. Une des difficultés de ce genre d'algorithme est l'utilisation importante de paramètres. Nous pensons tout d'abord qu'il pourrait être intéressant de développer une méthode d'élimination inverse pour gérer la liste tabou. Cette approche nous permettrait d'éliminer le paramètre sur la longueur de la liste tabou et définir le statut d'un mouvement de manière optimale. Nous pouvons également modifier les phases d'intensification et de diversification utilisées dans l'approche actuelle pour prendre plus en compte l'historique de la recherche (d'une façon similaire à la méthode proposée dans le chapitre 4 par exemple). Nous pensons finalement proposer un nouvel algorithme dans lequel le processus de recherche tabou serait appliqué par intermittence sur l'ensemble N des variables et sur l'ensemble N''. Cela permettrait par exemple d'intégrer la visite de solutions non réalisables au cours de la recherche.

Notons finalement qu'il est envisageable d'appliquer cette approche à d'autres problèmes d'optimisation. Une condition nécessaire à son bon fonctionnement est qu'on dispose d'un algorithme de programmation dynamique efficace pour le problème à résoudre.

Le processus d'intensification globale peut ainsi par exemple être adapté à la plupart des problèmes de la famille du sac-à-dos mentionnés dans le chapitre 1 de ce mémoire.

Recherche dispersée

Dans le chapitre 4 nous avons présenté une approche qui repose essentiellement sur la recherche dispersée. Nous avons montré que certains composants de l'algorithme ont une forte influence sur les performances globales du processus. Nous avons proposé l'intégration de phases d'intensification ainsi que l'utilisation de la mémoire. Cela permet d'avoir une nette amélioration de la qualité de la meilleure solution obtenue. L'utilisation de la méthode des chemins reliants pour accentuer les phases d'intensification est également bénéfique. Nous avons proposé un générateur de solutions permettant d'obtenir une population initiale élite. Les résultats finaux montrent que la qualité de la population initiale influe aussi sur la qualité des solutions finales de l'algorithme de recherche dispersée. Nous avons finalement montré que l'utilisation de solutions non réalisables au sein de la population des solutions peut être bénéfique pour le problème du sac-à-dos multidimensionnel. Notre algorithme rivalise avec un algorithme génétique performant pour ce problème. Les résultats en terme de qualité de la meilleure solution restent cependant inférieurs à ceux du chapitre 5 de cette thèse pour les problèmes de grande taille.

Plusieurs extensions de ce travail sont possibles. La première d'entre elles est liée aux expériences numériques et consiste à proposer un algorithme permettant un réglage « dynamique » des tailles des sous-populations au sein de la population référence. Ce réglage semble en effet difficile à réaliser de manière théorique d'après nos expériences numériques. Nous pouvons par contre modifier les tailles des sous-populations pendant la résolution en fonction de la mémoire de la recherche et de son évolution. Une seconde perspective est de combiner la recherche dispersée avec la recherche tabou. Nous avons commencé cela par l'introduction de l'intensification et de la mémoire. Nous pouvons cependant développer ce point en utilisant la mémoire dans les générateurs de solutions diverses par exemple et dans les combinaisons. L'hybridation avec d'autres méthodes est elle aussi possible en particulier avec les approches présentées dans le chapitre 5. Nous pouvons par exemple définir un générateur à partir des heuristiques du chapitre 5 pour produire des populations initiales élites et ensuite appliquer l'algorithme de recherche dispersée à partir de ces populations.

L'algorithme de recherche dispersée que nous avons proposé a été développé de manière à être le plus efficace possible pour le problème du sac-à-dos multidimensionnel. Il

n'en reste cependant pas moins adaptable à d'autres problèmes d'optimisation. La structure générale de l'algorithme et en particulier la génération d'une population diverse, la construction et le maintien de la population référence, la génération des sous-ensembles de solutions, sont réutilisables pour résoudre les problèmes en variables 0-1. Les processus d'amélioration des solutions et de combinaison peuvent aussi être repris ou facilement adaptés en fonction du problème traité.

Heuristiques convergentes basées sur des relaxations

Le travail présenté dans le chapitre 5 est celui qui permet d'obtenir les meilleures solutions pour le problème du sac-à-dos multidimensionnel. Les algorithmes proposés sont en contre-partie les plus coûteux mais restent dans des proportions raisonnables par rapport aux algorithmes permettant d'obtenir les meilleures valeurs pour ce problème. La mise en évidence du phénomène de dominance dans l'algorithme initial permet une réelle diminution du nombre de problèmes réduits à résoudre de manière exacte. Sa mise en pratique ne permet cependant pas toujours d'apporter un gain en temps d'exécution très important et son utilisation est plutôt intéressante lors du traitement des instances de grande taille. Les heuristiques basées sur des relaxations qui utilisent conjointement la relaxation en continu et la relaxation en nombres entiers mixtes permettent de définir un processus de résolution particulièrement robuste pour les problèmes en variables 0-1 mixtes. Même si la convergence de ces heuristiques n'est pas utilisable en pratique sur les instances de grande taille, elles permettent d'obtenir plusieurs nouvelles valeurs connues sur des instances corrélées et difficiles du problème du sac-à-dos multidimensionnel.

Nous avons quelques idées pour poursuivre le travail présenté dans ce chapitre. Nous pouvons dans un premier temps proposer un algorithme parallèle de l'heuristique *HIRI*. Elle consiste à utiliser un processeur pour chaque relaxation. Il faudrait alors mettre en place des mécanismes de mise à jour des bornes supérieures et inférieures entre les deux machines. Une autre perspective intéressante de ce travail se situe dans le développement d'heuristiques hybrides intégrant de la mémoire comme dans les algorithmes de recherche tabou. L'idée consiste à guider la recherche au cours des itérations dans les problèmes réduits. Ceux-ci ne seraient plus déterminés uniquement en fonction d'une solution optimale d'une relaxation mais aussi en fonction de l'historique de la recherche.

Les heuristiques basées sur des relaxations sont les algorithmes présentés dans ce mémoire les plus facilement applicables à d'autres problèmes d'optimisation. Le

fonctionnement même de ces heuristiques permet d'affirmer qu'elles peuvent être appliquées à la fois aux problèmes en variables 0-1 et aux problèmes en variables 0-1 mixtes grâce à l'introduction de la relaxation en nombres entiers mixtes.

Bibliographie

[AEOP02] R. K. Ahuja, O. Ergun, J.B. Orlin et A.P. Punnen. Survey of Very Large-Scale Neighborhood Search Techniques. *Discrete Applied Mathematics* 123:75–102, 2002.

[AF75] J. H. Ahrens et G. Finke. Merging and sorting applied to the zero-one knapsack problem. Operations Research, 23(6):1099-1109, 1975.

[AJ94] R. Aboudi et K. Jörnsten. Tabu search for general zero-one integer programs using the pivot and complement heuristic. ORSA Journal of Computing, 6:82-93, 1994.

[ASG04] I. Alaya, C. Solnon et K. Ghedira. Des fourmis pour le sac-à-dos multidimensionnel. Dans : 4èmes Journées Francophones de Recherche Opérationnelle (Francoro 2004), Fribourg, Suisse, pp. 159-160, 2004.

[Bal75] E. Balas. Facets of the knapsack polytope. Mathematical Programming, 8:146-164, 1975.

[BCDMP01] E. Balas, S. Ceria, M. Dawande, F. Margot et G. Pataki. OCTANE: A new heuristic for pure 0–1 programs. Operations Research, 49:207–225, 2001.

[BD02] D. Bertsimas et R. Demir. An approximate dynamic programming approach to multidimensional knapsack problems. Management Science, 48(4):550–565, 2002.

[Bea90] J. E. Beasley. OR-Library: distributing test problems by electronic mail. Journal of the Operational Research Society, 41(11):1069-1072, 1990. http://people.brunel.ac.uk/~mastjjb/jeb/info.html

[BEB06] V. Boyer, M. Elkihel et D. El Baz. Efficient heuristic for the 0-1 multidimensional knapsack problem. Actes de la 7ème conférence de la Société Française de Recherche Opérationnelle et d'Aide à la Décision, pp.95-106, 2006.

[Bel57] R. Bellman. Dynamic Programming. Princeton University Press, Princeton, 1957.

[BJ72] E. Balas et R. Jeroslow. Canonical cuts on the unit hypercube. SIAM Journal on Applied Mathematics, 23(1): 61-69, 1972.

[BM80] E. Balas et C. H. Martin. Pivot and complement–A heuristic for 0–1 programming. Management Sciences, 26:86–96, 1980.

[BMM01] G. J. Beaujon, S. P. Marin et G. C. McDonald. Balancing and Optimizing a Portfolio of R&D projects. Naval research Logistics, 48, pp 18 – 40, 2001.

[Bro79] J. R. Brown. The knapsack sharing problem. Operations Research, 27:341-355, 1979.

[BSW04] E. Balas, S. Schmieta et C. Wallace. Pivot and shift – a mixed integer programming heuristic. Discrete Optimization, 1:3-12, 2004.

[BT94] R. Battiti et G. Tecchiolli. The reactive tabu search. ORSA Journal on Computing, 6(2): 126-140, 1994.

[BYFA01] S. Balev, N. Yanev, A. Fréville et R. Andonov. A dynamic programming based reduction procedure for the multidimensional 0-1 knapsack problem. Rapport

de recherche, Université de Valenciennes, 2001.

[BZ80] E. Balas et E. Zemel. An algorithm for large zero-one knapsack problems. Operations Research, 28:1130-1154, 1980.

[Cab70] A. V. Cabot. An enumeration algorithm for knapsack problems. Operations Research, 18:306-311, 1970.

[Cap99] P. Cappanera. Discrete Facility Location and Routing of Obnoxious Facilities. Ph.D. thesis, Université de Milan (Italie). Disponible à : http://www.di.unipi.it/~cappaner.

[CB98] P. Chu et J. Beasley. A genetic algorithm for the multidimensional knapsack problem. Journal of Heuristics, 4: 63-86, 1998.

[CGLM01] V. Campos, F. Glover, M. Laguna et R. Martí. An Experimental Evaluation of a Scatter Search for the Linear Ordering Problem. Journal of Global Optimization, vol. 21, pp. 397-414, 2001.

[CH93] I. Charon et O. Hudry. The noising method: a new method for combinatorial Optimization. Operations Research Letters, 14: 133-137, 1993.

[CM94] Y. Crama et J. Mazzola. On the strength of relaxations of multidimensional knapsack problems, INFOR 32:219–225, 1994.

[CMM01] P. Chardaire, G. P. McKeown et J. A. Maki. Application of GRASP to the Multiconstraint Knapsack Problem. Lecture Notes In Computer Science; Actes de : EvoWorkshops on Applications of Evolutionary Computing, Vol. 2037, 30-39, 2001.

[CMMT96] V. D. Cung, T. Mautor, P. Michelon et A. Tavares. A scatter search based approach for the quadratic assignment problem. Rapport de recherche n°96/037. Laboratoire PRISM, Université de Versailles, 1996.

[CPT99] A. Caprara, D. Pisinger et P. Toth. Exact solution of the quadratic knapsack problem. INFORMS Journal on Computing, 11:125-137, 1999.

[CT05] P. Cappanera et M. Trubian. A Local Search Based Heuristic for the Demand Constrained Multidimensional Knapsack Problem, INFORMS JOC, 17(1), pp 82-98, 2005.

[Dan57] G. B. Dantzig. Discrete variable extremum problems. Operations Research, 5:266-277, 1957.

[DC99] M. Dorigo et G. Di Caro. The ant colony optimization metaheuristic. Dans : D. Corne, M. Dorigo et F. Glover (éditeurs), New ideas in optimization, McCrow-Hill, 11-32, 1999.

[DKW84] M. E. Dyer, N. Kayal et J. Walker. A branch and bound algorithm for solving the multiple choice knapsack problem. Journal of Computational and Applied Mathematics, 11:231-249, 1984.

[Dre88] A. Drexl. A simulated annealing approach to the multiconstraint zero–one knapsack problem. Computing, 40:1–8, 1988.

[DRP05] E. Danna, E. Rothberg et C. L. Pape. Exploring relaxations induced neighborhoods to improve MIP solutions. Mathematical Programming, 102(A):71-90, 2005.

[DRW95] M. E. Dyer, W. O. Riha et J. Walker. A hybrid dynamic programming/branch-and-bound algorithm for the multiple-choice knapsack problem. European Journal of Operational Research, 58:43-54, 1995.

[DS90] G. Dueck et T. Scheuer. Threshold accepting: A general purpose optimization algorithm. Journal of Computational Physics, 90:161–175, 1990.

[DV93] F. Dammeyer et S. Voss. Dynamic tabu list management using reverse elimination method. Annals of Operations Research 31–46, 1993.

[Ebe96] M. Eben-Chaime. Parametric solution for linear bicriteria knapsack models.

Management Science, 42:1565-1575, 1996.

[EC71] S. Eilon et N. Christofides. The loading problem. Management Science, 17:259-268, 1971.

[EG02] M. Ehrgott et X. Gandibleux (éditeurs). Multiple Criteria Optimization: State of the Art Annotated Bibliographical Surveys. Kluwer, 2002.

[FC84] A. M. Frieze et M. R. B. Clarke. Approximation algorithms for the m-dimensional 0-1 knapsack problem: worst-case and probabilistic analyses. European Journal of Operational Research, 15:100-109, 1984.

[FH05] A. Fréville et S. Hanafi. The multidimensional 0-1 knapsack problem – Bounds and computational aspects. Advances in Operations Research, Annals of Operations Research, M. Guignard and K. Spielberg (éditeurs), 139(1):195-227, 2005.

[Fid05] S. Fidanova. Ant Colony Optimization for Multiple Knapsack Problem and Model Bias, NAA'04. Lecture Notes in Computer Sciences, Springer, Germany, pages 282-289, 2005.

[Fis81] M. L. Fischer. The Lagrangean relaxation method for solving integer programming problems. Management Sciences, 27:1–18, 1981.

[FL03] M. Fischetti et A. Lodi. Local Branching. Mathematical Programming, 98(B): 23-47, 2003.

[FP82] D. Fayard et G. Plateau. An algorithm for the solution of the 0-1 knapsack problem. Computing, 28:269-287, 1982.

[FP86] A. Fréville et G. Plateau. Heuristics and reduction methods for multiple constraints 0–1 linear programming problems. European Journal of Operational Research, 24:206– 215, 1986.

[FP93a] A. Fréville et G. Plateau. Sac-à-dos multidimensionel en variables 0–1: Encadrement de la somme des variables à l'optimum. RAIRO Operations Research, 27:169-187, 1993.

[FP93b] A. Fréville et G. Plateau. An exact search for the solution of the surrogate dual for the bidmensional knapsack problem. European Journal of Operational Research,68:413-421, 1993.

[FP94] A. Fréville et G. Plateau. An efficient preprocessing procedure for the multidimensional 0-1 knapsack problem, Discrete Applied Mathematics, 2(2): 147-167, 1994.

[FP96] A. Fréville et G. Plateau. The 0-1 bidimensional knapsack problem: towards an efficient high-level primitive tool. Journal of Heuristics,2:147-167, 1996.

[FR89] T. A. Feo et M. G. C. Resende. A probabilistic heuristic for a computationally difficult set covering problem. Operations Research Letters, 8: 67-71, 1989.

[Fré04] A. Fréville. The multidimensional 0–1 knapsack problem: An overview. European Journal of Operational Research, 155 (1): 1-21, 2004.

[GF00] X. Gandibleux et A. Fréville. Tabu search based procedure for solving the 0-1 multi-objective knapsack problem: the two objective case. Journal of Heuristics, 6:361-383, 2000.

[GG66] P. C. Gilmore et R. E. Gomory. The theory and computation of knapsack functions. Operations Research, 14:1045-1075, 1966.

[GH02] F. Glover et S. Hanafi. Tabu search and finite convergence. Discrete Applied Mathematics, 119:3-36, 2002.

[GHS80] G. Gallo, P. L. Hammer et B. Simeone. Quadratic knapsack problems. Mathematical Programming Study, 12:132-149, 1980.

[GJ79] M. R. Garey et D. S. Johnson. Computers and Intractability: a Guide to the Theory of NP-Completeness. Freeman, 1979.

[GK96] F. Glover, et G. Kochenberger. Critical Event tabu Search for Multidimensional Knapsack Problems. Dans I.H. Osman et J.P. Kelly (éditeurs), Meta Heuristics: Theory and Applications, Kluwer Academic Publishers, pp 407 – 427, 1996.

[GKM99] V. Gabrel, A. Knippel et M. Minoux. Exact solutions of multicommodity network optimization problems with general step cost functions. Operations Research Letters, 25:15–23, 1999.

[GL79] G. V. Gens et E. V. Levner. Computational complexity of approximation algorithms for combinatorial problems. In Mathematical Foundations of Computer Science, pages 292-300, 1979.

[GL97] F. Glover et M. Laguna. Tabu Search. Kluwer Academic Publishers, 1997.

[Glo63] F. Glover. Parametric combinations of local job shop rules. Chapter IV, ONR Research Memorandum no. 117, GSIA, Carnegie Mellon University, Pittsburgh, PA, 1963.

[Glo65] F. Glover. A multiphase-dual algorithm for the zero–one integer programming problem. Operations Research, 13:879–919, 1965.

[Glo68] F. Glover. Surrogate constraints. Operations Research, 16:741–749, 1968.

[Glo77] F. Glover. Heuristics for integer programming using surrogate constraints. Decision Sciences, Vol.8, No1:156-166, 1977.

[Glo86] F. Glover. Future paths for integer programming and links to artificial intelligence. Computer and Operations Research, 13:533-549, 1986.

[Glo89] F. Glover. Tabu Search - Part I. ORSA, Journal on Computing, 1(3):190-206, 1989.

[Glo90] F. Glover. Tabu Search - Part II. ORSA, Journal on Computing, 2(1):4-32, 1990.

[Glo95] F. Glover. Scatter Search and Star Paths: Beyond the Genetic Metaphor. OR Spektrum, 17:125-137, 1995.

[Glo98] F. Glover. A Template for Scatter Search and Path Relinking. Dans : Hao, J.K., Lutton, E., Ronald, E., Schoenauer, M., Snyers D. (éditeurs). Artificial Evolution. Lecture Note in Computer Science 1363, Springer, pp. 13-54, 1998.

[Glo05] F. Glover. Adaptive Memory Projection Methods for Integer Programming. Dans : *Metaheuristic Optimization Via Memory and Evolution*, C. Rego and B. Alidaee (éditeurs), Kluwer Academic Publishers, pp. 425-440, 2005.

[GLW00] F. Glover, A. Lokketangen et D. L. Woodruff. Scatter Search to Generate Diverse MIP Solutions. Dans : OR Computing Tools for Modeling, Optimization and Simulation: Interfaces in Computer Science and Operations Research, Laguna, M., González-Velarde, J.L. (éditeurs), Kluwer Academic Publishers, pp. 299-317, 2000.

[GMF97] X. Gandibleux, N. Mezdaoui et A. Fréville. A tabu search procedure to solve multi-objective combinatorial optimization problems. Dans : R. Caballero, F. Ruiz et R. Steuer (éditeurs), Advances in Multiple Objective and Goal Programming, volume 455 of Lecture Notes in Economics and Mathematical Systems, pp. 291-300. Springer, 1997.

[GMK01] X. Gandibleux, H. Morita et N. Katoh. The supported solutions used as a genetic information in a population heuristic. Dans : First International Conference on Evolutionary Multi-Criterion Optimization, volume 1993 of Lecture Notes in Computer Science, pp. 429-442. Springer, 2001.

[GP70] H. Greenberg et W. Pierskalla. Surrogate mathematical programs. Operations Research, 18:924–939, 1970.

[GP82] B. Gavish et H. Pirkul. Allocation of databases and processors in a distributed data processing. Dans : J. Akola (éditeur), Management of Distributed Data

Processing, North-Holland, Amsterdam, pp. 215–231, 1982.

[GP85] B. Gavish et H. Pirkul. Efficient algorithms for solving multiconstraint zero-one knapsack problems to optimality. Mathematical Programming, 31:78-105, 1985.

[GP99] C. Guéret et C. Prins. A new lower bound for the open-shop problem. Annals of Operations Research, 92:165-183, 1999.

[Gre03] P. Greistorfer. A Tabu Scatter Search Metaheuristic for the Arc Routing Problem. Computers & Industrial Engineering, 44:249-266, 2003.

[Han86] P. Hansen. The steepest ascent mildest descent heuristic for combinatorial programming. In: Congress on Numerical Methods in Combinatorial Optimization, Capri, Italie, 1986.

[Han01] S. Hanafi. On the convergence of tabu search. Journal of Heuristics, 7:47-58, 2001.

[HF98] S. Hanafi et A. Fréville. An efficient tabu search approach for the 0–1 multidimensional knapsack problem. European Journal of Operational Research, 106:659–675, 1998.

[HF01] S. Hanafi et A. Fréville. Extension of reverse elimination method through a dynamic management of the tabu list. RAIRO Operations Research, 35:251–267, 2001.

[HFA96] S. Hanafi, A. Fréville et A. El Abdellaoui. Comparison of heuristics for the 0–1 multidimensional knapsack problem, in: I.H. Osman, J.P. Kelly (éditeurs), Meta-Heuristics: Theory and Applications, Kluwer Academic Publishers, pp.449–466, 1996.

[HHS05] M. Hifi, H. M'Halla et S. Sadfi. An exact algorithm for the knapsack sharing problem. Computers & Operations Research, 32:1311-1324, 2005.

[Hil69] F. S. Hillier. Efficient heuristic procedures for integer linear programming with an interior. Operations Research, 17:600–637, 1969.

[HL05] L. M. Hvattum et A. Lokketengen. Experiments using Scatter Search for the Multidemand Multidimensional Knapsack Problem. Actes de : the 6th Metaheuristics International Conference (MIC05), pp. 526-531, 2005.

[HM01] P. Hansen et N. Mladenovic. Variable neighborhood search: Principles and applications. European Journal of Operational Research, 130:449-467, 2001.

[HM05] M. Hifi et M. Michrafy. A reactive local search-based algorithm for the disjunctively constrained knapsack problem. Journal of the Operational Research Society, à paraître.

[HM06] M. Hifi et M. Michrafy. Reduction strategies and exact algorithms for the disjunctively constrained knapsack problem. Computers & Operations Research, à paraître.

[Hol75] J. H. Holland. Adaptation in Natural and Artificial Systems, Cambridge, MA : The M.I.T. Press, 1975.

[HR00] R. R. Hill et C. H. Reilly. The effects of coefficient correlation structure in two-dimensional knapsack problems on solution performance. Management Science, 46:302-317, 2000.

[HS02] M. Hifi et S. Sadfi. The knapsack sharing: an exact algorithm. Journal of Combinatorial Optimization, 6:34-54, 2002.

[HS74] E. Horowitz et S. Sahni. Computing partitions with applications to the knapsack problem. Journal of the ACM, 21:277-292, 1974.

[HSS02] M. Hifi, S. Sadfi et A. Sbihi. An efficient algorithm for the knapsack sharing problem. Computational Optimization and Applications, 23:27-45, 2002.

[HV98] C. Haul et S. Voss. Using surrogate constraints in genetic algorithms for solving multidimensional knapsack problems, in: D.L. Woodru. (éditeur), Advances in

Computational and Stochastic Optimization, Logic Programming, and Heuristic Search, Kluwer Academic Publishers, pp.235–251, 1998.

[IK75] O. H. Ibarra et C. E. Kim. Fast approximation algorithms for the knapsack and sum of subset problem. Journal of the ACM, 22:463-468, 1975.

[Ilo03] ILOG. CPLEX 9.0 (user's manual), 2003.

[KMW74] G. Kochenberger, G. McCarl et F. Wymann. A heuristic for general integer programming. Decision Sciences, 5:36–44, 1974.

[KP99] H. Kellerer et U. Pferschy. A new fully polynomial time approximation scheme for the knapsack problem. Journal of Combinatorial Optimization, 3:59-71, 1999.

[KPP04] H. Kellerer, U. Pferschy et D. Pisinger. Knapsack Problems. Springer, ISBN: 3-540-40286-1, 546 pages, 2004.

[KS81] B. Korte et R. Schrader. On the existence of fast approximation schemes. In O.L. Mangasarian, R. R. Meyer et S. M. Robinson (éditeurs), Nonlinear Programming, volume 4, pages 415-137. Academic Press, 1981.

[LA05] M. Laguna et V. Armentano. Lessons from Applying and Experimenting with Scatter Search. Dans : Metaheuristic Optimization Via Adaptive Memory and Evolution: Tabu Search and Scatter Search, C. Rego and B. Alidaee (éditeurs), Norwell, MA: Kluwer Academic Publishers, pp. 229-246, 2005.

[LG96] A. Lokketangen et F. Glover. Probabilistic move selection in tabu search for zero–one mixed integer programming problems. Dans : I.H. Osman, J.P. Kelly (éditeurs), Meta-Heuristics: Theory and Applications, Kluwer Academic Publishers, pages : 467–487, 1996.

[LG98] A. Lokketangen et F. Glover. Solving zero–one mixed integer programming problems using tabu search. Special Issue on Tabu Search. European Journal of Operational Research, 106:624–658, 1998.

[Lic05] D. Lichtenberger. An extended local branching framework and its application to the multidimensional knapsack problem. Master's thesis, Vienna University of Technology, Institute of Computer Graphics and Algorithms, March 2005.

[LJS94] A. Lokketangen, K. Jörnsten et S. Storoy. Tabu search within a pivot and complement framework. International Transactions of Operational Research, 1:305-316, 1994.

[LM03] M. Laguna et R. Martí. Scatter Search. Methodology and Implementations in C. Kluwer Academic Publishers, 2003.

[LM79] R. Loulou et E. Michaelides. New greedy-like heuristics for the multidimensional 0–1 knapsack problem. Operations Research, 27:1101–1114, 1979.

[LS55] J. H. Lorie et L. J. Savage. Three problems in capital rationing. The Journal of Business, 28:229-239, 1955.

[Mat97] G. B. Mathews. On the partition of numbers. Actes de : the London Mathematical Society, 28:486-490, 1897.

[MCC97] D. Romero-Morales, E. Carrizosa et E. Conde. Semi-Obnoxious Location Models: a Global Optimization Approach. European Journal of Operational Research, 102, pp 295 – 301, 1997.

[MCS01] H. Meier, N. Christofides et G. Salkin. Capital budgeting under uncertainty—an integrated approach using contingent claims analysis and integer programming. Operations Research, 49:196–206, 2001.

[MDW05] R. J. Moraga, G. W. DePuy et G. E. Whitehouse. Meta-RaPS approach for the 0-1 Multidimensional Knapsack Problem. Computers & Industrial Engineering, 48: 83–96, 2005.

[MH78] R. Merkle et M. Hellman. Hiding information and signatures in trapdoor knapsacks. IEEE Transactions on Information Theory, IT-24:525-530, 1978.

[Mit70] L. G. Mitten. Branch-and-Bound Methods: General Formulation and Properties. Operations Research, vol.18, pages 24-34, 1970.

[MJS97] M. Moser, D. P. Jokanovic et N. Shiratori. An algorithm for the multidimensional multiple-choice knapsack problem. IEICE Transactions A, E80:582-589, 1997.

[MLL00] R. Martí, H. Lourenço et M. Laguna. Assigning Proctors to Exams with Scatter Search. Dans : Laguna, M., González-Velarde, J. L. (éditeurs), Computing Tools for Modeling, Optimization and Simulation: Interfaces in Computer Science and Operations Research, Kluwer Academic Publishers, Boston, pp. 215-227, 2000.

[MM57] A. S. Manne et H. M. Markowitz. On the solution of discrete programming problems. Econometrica, 25:84–110, 1957.

[MM78] R. E. Marsten, T. L. Morin. A hybrid approach to discrete mathematical programming, Mathematical Programming, 14: 21–40, 1978.

[MO84] M.J. Magazine et O. Oguz. A heuristic algorithm for the multidimensional zero–one knapsack problem. European Journal of Operational Research, 16:319–326, 1984.

[MS89] R. E. Marsten et J. Singhal. XMP (Experimental Mathematical Programmiung), XMP Software, Inc., Tucson, Arizona, 1989.

[MT77] S. Martello et P. Toth. An upper bound for the zero-one knapsack problem and a branch and bound algorithm. European Journal of Operational Research, 1:169-175, 1977.

[MT81] S. Martello et P. Toth. A bound and bound algorithm for the zero-one multiple knapsack problem. Discrete Applied Mathematics, 3:275-288, 1981.

[MT84] S. Martello et P. Toth. A mixture of dynamic programming and branch-and-bound for the subset-sum problem. Management Science, 30:765-771, 1984.

[MT90] S. Martello et P. Toth. Knapsack Problems: Algorithms and Computer Implementations. J. Wiley, 1990.

[MT03] S. Martello et P. Toth. An exact algorithm for the two-constraint 0-1 knapsack problem. Operations Research, Vol.51, Numéro 5, 826-835, 2003.

[Nau78] R. Nauss. The zero-one knapsack problem with multiple-choice constraints. European Journal of Operational Research, 2:125-131, 1978.

[NE01] M. Nediak et J. Eckstein. Pivot, cut and dive: A heuristic for 0–1 mixed integer programming, Rutcor research report, Rutgers University, USA, October 2001.

[NU69] G. L. Nemhauser et Z. Ulmann. Discrete dynamic programming and capital allocation. Management Science, 15:494-505, 1969.

[NW99] G. L. Nemhauser et L. A. Wolsey. Integer and combinatorial optimization. Willey-Interscience, New York, 1999.

[OL96] I. H. Osman et G. Laporte. Metaheuristics: a bibliography. Annals of Operations Research, 63: 513-623, 1996.

[Ped04] Joao Pedro Pedroso. Tabu search for mixed integer programming. Rapport de recherche DCC-2004-02. Université de Porto (Portugal), 2004.

[PHD02] R. Parra-Hernandez et N. Dimopoulos. A new heuristic for solving the multi-choice multidimensional knapsack problem. Rapport de recherche, Department of Electrical and Computer Engineering, University of Victoria, 2002.

[Pir02] M. Pirlot. Métaheuritiques pour l'optimisation combinatoire : un aperçu général. Dans : J. Teghem et M. Pirlot (éditeurs), Optimisation approchée en recherche opérationnelle – Recherches locales, réseaux neuronaux et

satisfaction de contraintes, pages 25-55. Hermès Science Publications, Paris, 2002.

[Pir87] H. Pirkul. A heuristic solution procedure for the multiconstraint zero–one knapsack problem. Naval Research Logistics, 34:161–172, 1987.

[Pis95a] D. Pisinger. An expanding-core algorithm for the exact 0-1 knapsack problem. European Journal of Operational Research, 87:175-187, 1995.

[Pis95b] D. Pisinger. A minimal algorithm for the multiple-choice knapsack problem. European Journal of Operational Research, 83:394-410, 1995.

[Pis99a] D. Pisinger. An exact algorithm for large multiple knapsack problems. European Journal of Operational Research, 114:528-541, 1999.

[Pis99b] D. Pisinger. Core problems in Knapsack Algorithms. Operations Research, 47: 570-575, 1999.

[Pla01] F. Plastria. Static Competitive Facility Location: an overview of Optimization Approaches. European Journal of Operational Research, l 129, pp 461-470, 2001.

[PRG05] J. Puchinger, G. R. Raidl et M. Gruber. Cooperating Memetic and Branch-and-Cut Algorithms for Solving the Multidimensional Knapsack Problem. Actes de : the 6th Metaheuristics International Conference, MIC2005, Vienne (Autriche), 2005.

[PTT02] A. Plateau, D. Tachat et P. Tolla. A hybrid search combining interior point methods and metaheuristics for 0–1 programming. International Transactions in Operational Research, 9:731–746, 2002.

[RL01] T. K. Ralphs et L. Ladanyi. COIN/BCP User's Manual. http://www.coin-or.org/Presentations/bcp-man.pdf, 2001.

[RS89] M. J. Rosenblatt et Z. Sinuany-Stern. Generating the discrete efficient frontier to the capital budgeting problem. Operations Research, 37:740-747, 1989.

[Sal70] H. M. Salkin. On the merit of the generalized origin and restarts in implicit enumeration. Operations Research, 18:549-554, 1970.

[Sav94] M. W. P. Savelsbergh. Preprocessing and Probing Techniques for Mixed Integer Programming Problems. ORSA Journal of Computing 6(4):445--454, 1994.

[Sch85] J. D. Schaffer. Multiple objective optimization with vector evaluated genetic algorithms. Dans : Genetic Algorithms and their Applications. Actes de : the First International Conference on Genetic Algorithms, pp. 93-100, 1985.

[SCF06a] C. G. da Silva, J. Clímaco et J. Figueira. A scatter search method for bi-criteria {0, 1}-knapsack problems. European Journal of Operational Research, Volume 169, Issue 2, 1: 373-391, 2006.

[SCF06b] C. G. da Silva, J. Clímaco et J. Figueira. A Scatter Search Method for the Bi-Criteria Multi-Dimensional {0,1}-Knapsack Problem using Surrogate Relaxation. A paraître dans : Journal of Mathematical Modelling and Algorithms, 2006.

[Sha98] P. Shaw. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. Lecture Notes In: Computer Science, Vol. 1520. Actes de : the 4th International Conference on Principles and Practice of Constraint Programming, 417 – 431, 1998.

[Shi79] W. Shih. A branch and bound method for the multiconstraint zero-one knapsack problem. Journal of the Operational Research Society, 30:369-378, 1979.

[Sko02] G. Skorobohatyj. MP-TESTDATA: Integer/mixed-integer programming problems, 2002. http://elib.zib.de/pub/Packages/mp-testdata/ip//index.html

[SLS78] A. L. Soyster, B. Lev et W. Slivka. Zero-one programming with many variables and few constraints. European Journal of Operational Research, 2: 195-201, 1978.

[SS06] K. Sörensen et M. Sevaux. MA|PM: memetic algorithms with population management. Computers & Operations Research, Volume 33, Issue 5: 1214-1225, 2006.

[ST68] S. Senju et Y. Toyoda. An approach to linear programming with 0-1 variables. Management Science, 15:196-207, 1968.

[SZ79] A. Sinha et A. A. Zoltners. The multiple-choice knapsack problem. Operations Research, 27:503-515, 1979.

[The73] A. Thesen. Scheduling of computer programs in a multiprogramming environment. BIT, 13:208-216, 1973.

[The75] A. Thesen. A recursive branch and bound algorithm for the multidimensional knapsack problem. Naval Research Logistics Quarterly, 22:341-353, 1975.

[THK80] F. Tillman, C. Hwang et W. Kuo. Optimisation of System Reliability. Marcel Dekker (éditeurs), New York, 1980.

[TNP03] B. Thiongane, A. Nagih et G. Plateau. Lagrangean heuristics combined with reoptimization for the 0-1 biknapsack problem. Computer Science Laboratory, University of Paris-Nord, 2003.

[Tot80] P. Toth. Dynamic programming algorithms for the zero-one knapsack problem. Computing, 25:29-45, 1980.

[Toy75] Y. Toyoda. A simplified algorithm for obtaining approximate solutions to zero–one programming problems. Management Sciences, 21:1417–1427, 1975.

[TV94] J. Thiel et S. Voss. Some experiences on solving multiconstraint zero–one knapsack problems with genetic algorithms. INFOR 32, 226–242, 1994.

[UT97] E. L. Ulungu et J. Teghem. Solving multi-objective knapsack problems by a branch-and-bound procedure. Dans : J. Climacao (éditeur), Multicriteria Analysis, pp. 269-278. Springer, 1997.

[VH01a] M. Vasquez et J. K. Hao. A logic-constrained knapsack formulation and a tabu algorithm for the daily photograph scheduling of an earth observation satellite. Computational Optimization and Applications, 20:137–157, 2001.

[VH01b] M. Vasquez et J. K. Hao. Une approche hybride pour le sac-à-dos multidimensionnel en variables 0–1. RAIRO Operations Research, 35:415–438, 2001.

[Via98] F. Viader. Méthodes de Programmation Dynamique et de Recherche Arborescente pour l'Optimisation Combinatoire : Utilisation Conjointe des deux approches et Parallélisation d'Algorithmes. Thèse de doctorat, Université Paul Sabatier, Toulouse, France, 1998.

[VV05] M. Vasquez et Y. Vimont. Improved results on the 0-1 multidimensional knapsack problem. European Journal of Operational Research, 165: 70-81, 2005.

[WHFB06] C. Wilbaut, S. Hanafi, A. Fréville et S. Balev. Tabu Search: Global Intensification using Dynamic Programming. Control and Cybernetic, Vol.35, Numéro 3, 2006.

[WN67] H. M. Weingartner et D. N. Ness. Methods for the solution of the multidimensional 0/1 knapsack problem. Operations Research, 15:83-103, 1967.

[Wol75] L. A. Wolsey. Factes of linear inequalities in 0-1 variables. Mathematical Programming, 8:165-178, 1975.

[YFK98] T. Yamada, M. Futakawa et S. Kataoka. Some exact algorithms for the

knapsack sharing problem. European Journal of Operational Research, 106:177-183, 1998.

[YK01] T. Yamada et S. Kataoka. Heuristic and exact algorithms for the disjunctively constrained knapsack problem. EURO 2001, Rotterdam, Pays-Bas, Juillet 2001.

[YKW02] T. Yamada, S. Kataoka et K. Watanabe. Heuristic and Exact Algorithms for the Disjunctively Constrained Knapsack Problem. Information Processing Society of Japan Journal, Vol. 43, No.9:2864-2870, 2002.

[Yu96] G. Yu. On the max-min 0-1 knapsack problem with robust optimization applications. Operations Research, 44:407-415, 1996.

[ZO04] C. W. Zhang et H. L. Ong. Solving the biobjective zero–one knapsack problem by an efficient LP-based heuristic. European Journal of Operational Research, Vol.159(3): 545-557, 2004.