UNIVERSITÉ DE NANTES

ULB UNIVERSITÉ LIBRE DE BRUXELLES

Année académique 2020-2021

Travail d'Étude et de Recherche

Ensembles bornants

Auteurs :

Lucas BAUSSAY
Mathilde HAVARD-SEDENO
Jules NICOLAS-THOUVENIN

Encadrants :

Pr. Xavier GANDIBLEUX
Pr. Anthony PRZYBYLSKI

Ecoles :

UNIVERSITÉ de NANTES - M1 ORO
UNIVERSITÉ LIBRE de BRUXELLES - MA1

Rapporteur :

Pr. Alexandre GOLDSZTEJN

`https://github.com/LucasBaussay/BranchAndBoundBiObjKp`

Abstract

Français

Ce travail de recherche s'inscrit dans le cadre des problèmes d'optimisation multi-objectifs et porte particulièrement sur l'usage d'ensembles bornants. Nous étudions tout d'abord le paradigme des problèmes multi-objectifs et ses spécificités, pour ensuite étudier l'application d'algorithmes de résolution sur le problème du sac-à-dos bi-objectifs mondimensionnel. Nous traiteront en profondeur le cas du branch-and-bound. Nous commençons par un algorithme utilisant comme ensemble bornant l'enveloppe convexe de l'ensemble des solution, obtenue avec un solver général (JuMP). Puis nous allons pouvoir comparer cette résolution avec différents améliorations allant de l'utilisation de la relaxation linéaire dans la dichotomie à l'implémentation d'une heuristique primale. Suite à l'études de tout ces algorithmes, on pourra observer de bons résultats pour l'un d'entre eux, ne pouvant tout de même pas dépasser les résultats d'un algorithme ϵ-contrainte.

English

In this research work, we study the paradigm of multi-objective optimization and focuses on the use of upper and lower bound sets. We first study multi-objective problems and their specificity, and then study the application of algorithms on the mono-dimensional bi-objective knapsack problem, specializing on the branch-and-bound. First we start with an algorithm using as upper-bound set the convex envelope of the solution set, obtained with a general solver (JuMP). Then we will be able to compare this algorithm with different improvements, ranging from the use of linear relaxation in the dichotomy to implementing a primal heuristic. Following the study of all these algorithms, we will be able to observe good results for one of them, despite not being able to exceed the results of the ϵ-constraiint method.

Remerciements

Tout d'abord, nous tenons à remercier nos encadrants, les professeurs Xavier Gandibleux et Anthony Pzyblyski. Ils ont su nous initier aux toutes les notions nécessaires et nous ont accompagné au long de notre travail. Leurs avis et idées ont permis d'orienter nos travaux et de construire cette étude.

Nous remercions d'avance le professeur Alexandre Goldsztejn pour le temps qu'il consacrera à la lecture de ce travail et les conseils qu'il pourrait être amené à nous donner.

Rien de cela n'aurait pu être possible sans la bonne humeur constante en cette période. Les discussions tardives dans la cuisine se sont avérées fructueuses dans le bon déroulement de ce travail de recherche.

Notations

Notation	Signification
$\mathbb{R}, \mathbb{N}$	Ensemble des nombres réels, ensemble des entiers positifs
$\mathbb{R}^n$	Ensemble de vecteurs de n nombres réels
n, p, m	Nombre de variables, d'objectifs et de contraintes du problème
$x = (x_1, ..., x_n)$	Solution et vecteur de valeurs des variables
$z(x) = (z_1(x), ..., z_p(x))$	Vecteur des fonctions objectifs
$y = (y_1, ..., y_p)$	Point et vecteur de valeurs pour les fonctions objectifs
X	Ensemble des solutions réalisables
$Y = z(X)$	Image des solutions réalisables dans l'espace des objectifs
X_E	Ensemble complet de solutions efficaces
Y_N	Ensemble des points non dominés
X_{SE}	Ensemble des solutions efficaces supportées
Y_{SN}	Ensemble des points supportés non dominés
X_{SE1}	Ensemble des solutions extrêmes supportées efficaces
X_{SE2}	Ensemble des solutions non-extrêmes supportées et efficaces
Y_{SE1}	Ensemble des points extrêmes supportées non-dominés
Y_{SE2}	Ensemble des points non-extrêmes supportées et non-dominées
X_{NE}	Ensemble des solutions non-supportées efficaces
Y_{NN}	Ensemble des points non-supportées non-dominés
Y_N	Ensemble des points non-dominés d'un sous-problème
conv(S)	Enveloppe convexe de S
$\Delta\ (y^l, y^r)$	Triangle défini par les points y^r, y^l et (y_1^l, y_2^r)
cl(S)	Fermeture de l'ensemble S

Table des matières

Introduction

La recherche opérationnelle vise à déterminer la ou les meilleures décisions à prendre afin d'optimiser un système – depuis les choix de fabrication pour la création d'un produit à l'affectation de cargos sur un quai de déchargement. La recherche opérationnelle repose sur des méthodes scientifiques, essentiellement mathématiques et algorithmiques. Elle fait partie des disciplines d'aide à la décision. Ses applications sont diverses. L'optimisation peut concerner la maximisation des profits d'une entreprise, la minimisation des coûts de production ou des dégâts environnementaux. Des domaines d'applications spécifiques ont émergés depuis son essor au milieu de vingtième siècle, comme par exemple les problèmes d'ordonnancement, de tournées de véhicules ou de stabilité de réseaux énergétiques.

Un des problème les plus étudiés est le problème du sac-à-dos. Dans sa forme la plus simple – un objectif et une contrainte – un décideur dispose d'un certain nombre d'objets à placer dans un sac-à-dos. Celui-ci a une capacité limitée et chaque objet possède un poids et un profit. Le décideur se demande alors quels sont les objets à choisir afin de maximiser le profit total du sac. Ce problème est très étudié dans sa version mono-objectif car il se retrouve dans de nombreux problèmes d'optimisation, en tant que sous-problème. C'est donc un sujet très actif dans le domaine de la recherche. Il est également utilisé dans de multiples applications concrètes, comme les systèmes d'aide à la gestion de portefeuille ou la découpe de matériau.

Dans notre travail, nous nous pencherons sur le cas multi-objectifs. Comme son nom l'indique, un problème multi-objectif présente plusieurs fonctions objectifs. Le but consiste donc à trouver les solutions admissibles minimisant ou maximisant ces fonctions objectifs.

Tous ces points seront abordés dans la suite de notre travail. Il est donc important de d'abord définir ces notions. Ainsi, dans un premier temps, nous présenterons les fondamentaux du multi-objectifs. Les outils de comparaison de solutions seront redéfinis et la notion primordiale d'ensemble bornant sera explicitée. Nous nous pencherons ensuite sur les méthodes de résolution qui peuvent être appliquées dans ce cas. Plus précisément, nous étudierons la méthode de l'épsilon-contrainte (notée ϵ-contrainte) et celle du branch-and-bound. Étant donné que nous nous focalisons sur le problème du sac-à-dos, ce dernier sera introduit en détail. Nous présenterons ensuite le travail réalisé, qui consiste en l'implémentation d'un branch-and-bound appliqué à la résolution du problème du sac-à-dos en multi-objectifs. Ce travail a été réalisé en plusieurs étapes, résultat de différents obstacles et de multiples idées d'amélioration. Toutes ces réflexions seront explicitées. Nous reviendrons ensuite sur l'implémentation finale du code, ainsi que le manuel d'utilisation. Au terme de ce travail, des pistes d'amélioration seront entrevues et seront présentées en dernière partie.

La porte d'entrée du travail est l'étude des ensembles bornants. Au fur-et-à-mesure, notre objectif s'est affiné et s'est concentré sur l'algorithme de branch-and-bound. Le principal objectif est d'étudier l'efficacité d'un algorithme de branch-and-bound global à résoudre le problème du sac à dos mono-dimensionnel bi-objectifs. Et principalement de le comparer à un algorithme ϵ-contraintes en espérant trouver une formulation permettant de dépasser l'efficacité de ce dernier.

Étudier cette problématique nous a amené à étudier plusieurs pistes de recherche possibles. Nous avons donc fait certains choix concernant les algorithmes et les problèmes étudiés. Ces choix pourront être remis en questions plus tard.

Chapitre 1

Présentation et notions

1.1 Environnement de travail

Toutes les expérimentations qui seront présentées dans ce papier, ont été exécutées sur une machine virtuelle fournie par Google Cloud Compute :

Environnement matériel	
Processeur	1 sur 8 vCPU
Système d'exploitation	Debian 10
Mémoire vive	32Go

Environnement logiciel	
Language	Julia v1.4.2
Librairies	GLPK, JuMP, libCombo

Les graphes ont été réalisés sous Julia, grâce au package PyPlot.

Pour toutes les expérimentations qui suiveront, le même protocole a été mis en place, pour l'étude d'un algorithme :

- on souhaite étudier notre algorithme sur des problèmes variant de 50 à 150 objets (augmentant de 10 en 10 objets) ;
- pour chaque nombre d'objet, on va créer 5 instances aléatoires ;
- on résout donc les 5 instances avec notre algorithme et on récupère les moyennes des temps d'exécution ainsi que le nombre moyen de sous-problèmes rencontrés ;

On résout les 5 instances avec un budget limite de 10 minutes. On va donc en résoudre le plus possible dans le temps imparti.

Ce budget de calcul a été fixé à 10 minutes pour éviter des temps d'expérimentation trop long, la complexité des algorithmes étant exponentielle. De plus, dans le cas d'un problème réel, le nombre d'objets du sac à dos peut varier jusqu'à plusieurs milliers de variables. Dans ce cas, un budget de calcul de 10 minutes pour des instances ne pouvant aller que jusqu'à 150 variables semble tout à fait raisonnable.

1.2 Définitions

1.2.1 Multi-objectifs

Dans le cas mono-objectif, la définition d'un problème d'optimisation est :

$$\begin{bmatrix} min\ \mathrm{z(x)} \\ \mathrm{s.t.} \quad x \in X \end{bmatrix}$$

Dans le cas multi-objectifs, plusieurs fonctions objectifs sont considérées, la forme standard du problème (MOP) est donc :

$$\begin{bmatrix} min\ (\ \mathrm{z}_1\mathrm{(x)}, ..., \ \mathrm{z}_p\mathrm{(x)}\ ) \\ \mathrm{s.t.} \quad x \in X \end{bmatrix}$$

où $X \subseteq \mathbb{R}^n$ est l'ensemble de définition du problème. x désigne une solution et $y = z(x)$ le point associé à la solution.

Ce problème possède $p \in \mathbb{N}^*$ fonctions objectifs, et chaque fonction objectif z_i est de la forme : $z_i : X \mapsto \mathbb{R}$.

La principale différence entre mono et multi objectifs, réside dans la notion d'optimalité d'une solution. Quand il n'y a qu'un objectif, la relation d'ordre strict $\geq$ suffit à comparer les solutions entre elles. Mais à partir de deux objectifs, il n'existe pas de relation d'ordre stricte. Le principe d'un unique optimum ne peut donc pas exister. Pour pallier à cette différence, observons la notion de dominance dans un espace à plusieurs dimensions.

Dans un contexte multi-objectifs, il est nécessaire de définir de nouveaux critères de comparaison. On parle alors de relation de dominance entre les solutions, comme défini par Pareto. Ces relations ne permettent pas d'établir un ordre complet, mais de déterminer, lorsque cela est possible, si une solution est meilleure qu'une autre, c'est-à-dire qu'elle la domine.

Définition 1. (DOMINANCE DE PARETO). *Soit $y_1, y_2 \in \mathbb{R}^p$, avec $p \geq 2$.*

- *Le point y_1 domine strictement le point y_2 ($y^1 > y^2$) si*

$$\forall i \in \{1, ..., p\} \qquad y_1^i > y_2^i$$

- *Le point y_1 domine faiblement le point y_2 ($y_1 \geqq y_2$) si*

$$\forall i \in \{1, ..., p\} \qquad y_1^i \geq y_2^i$$

- *Le point y_1 domine le point y_2 ($y_1 \geq y_2$) si*

$$y_1 \geqq y_2$$
$$et\ \exists i \in \{1, ..., p\} \qquad y_1^i > y_2^i$$

Si les comparaisons décrites ci-dessus ne sont pas applicables à deux points, on dit qu'ils sont incomparables entre eux. A partir de ces relations de dominance, on peut définir la notion de solution efficace comme suit.

Définition 2. (SOLUTION EFFICACE ET POINTS NON-DOMINÉS). *Soit $\hat{x} \in X$.*

- *$\hat{x}$ est efficace s'il n'existe pas de $x \in X$ tel que $z(x) \geq z(\hat{x})$. $z(\hat{x})$ est alors dit non-dominé.*
- *$\hat{x}$ est dite faiblement efficace si il n'existe pas de $x \in X$ tel que $z(x) \geqq z(\hat{x})$. $z(\hat{x})$ est alors dit faiblement non-dominé.*

Par extension, on obtient l'ensemble des solutions efficaces et l'ensemble des points non-dominés.

Définition 3. (ENSEMBLE DE SOLUTIONS EFFICACES ET ENSEMBLE DE POINTS NON-DOMINÉS). *L'ensemble des solutions efficaces $X_E \subset X$ est l'ensemble $\{x \in X : \nexists\, x' \in X, z(x') \geq z(x)\}$. Son image dans l'espace des objectifs, i.e $z(X_E)$, est appelée ensemble non-dominé Y_N.*

On distingue donc l'espace $\mathbb{R}^n$ – qui correspond au domaine des variables de décision – de l'espace objectif $\mathbb{R}^p$ – qui comporte l'image des solutions.

L'ensemble des images des solutions efficaces sont des points extrêmes de l'enveloppe convexe (par abus de langage, on parle d'enveloppe convexe pour désigner la partie de l'enveloppe convexe de Y dessinée par les points extrêmes non dominés). Cette enveloppe convexe peut être obtenue par la scalarisation des fonctions objectifs du problème. Geoffrion [5] définit ce problème de scalarisation par une somme pondérée de la manière suivante :

Théorème 1. *(Geoffrion, 1968). Soit un problème d'optimisation multi-objectifs P, avec p fonctions objectifs. P_λ est un problème de somme pondérée avec λ le vecteur de poids :*

$$max\{\lambda_1\ z_1(x)\ + ... + \lambda_p z_p(x) :\ x \in X\} \tag{1.1}$$

Supposons que x est la solution optimale de P_λ. Alors :*

- *si $\lambda \in \mathbb{R}^p_{\geq}$, alors x* est faiblement efficace ;*
- *si $\lambda \in \mathbb{R}^p_{>}$, alors x* est efficace ;*
- *si $\lambda \in \mathbb{R}^p_{\geq}$ et x* est l'unique solution optimale de P_λ, alors x* est efficace ;*

Avec :

— $\mathbb{R}^p_{\geq} = \{x \in \mathbb{R}^p : x \geq 0_p\}$
— $\mathbb{R}^p_{>} = \{x \in \mathbb{R}^p : x > 0_p\}$
— $\mathbb{R}^p_{\geqq} = \{x \in \mathbb{R}^p : x \geqq 0_p\}$

Ainsi, s'il existe un vecteur $\lambda \in \mathbb{R}^p$ tel que x est une solution optimale du problème P_λ, cette solution x est dite supportée. Par extension, on définit X_{SE} comme l'ensemble des solutions supportées efficaces et Y_{SN} - qui équivaut à $z(X_{SE})$ - comme l'ensemble des points supportés non-dominés. Chaque élément de ce dernier est situé sur l'enveloppe convexe Y.

À l'inverse, une solution efficace non-supportée est un solution efficace x pour laquelle il n'existe aucun $\lambda \in \mathbb{R}^p$ avec x une solution optimale pour P_λ. L'ensemble de ces solutions non-supportées efficaces est X_{NE}. Son image dans l'espace des objectifs est Y_{NN} et correspond donc aux points non-dominés non-supportés. Ce sont ces points efficaces qui seront les plus difficiles à trouver et qui apporteront toute la complexité aux algorithmes.

Pour ce qui est de l'ensemble des solutions supportées efficaces X_{SE}, on distingue deux cas. Si l'image du point correspond à un point extrême de l'enveloppe convexe de Y, on dit de lui qu'il est extrême – c'est l'ensemble X_{SE1}. Réciproquement, si l'image d'un point supporté efficace n'est pas un point extrême de $conv\,(Y)$, on dit qu'il est non-extrême – et on note l'ensemble de ces points X_{SE2}. Cela permet de définir deux nouveaux ensembles dans l'espace des objectifs : Y_{SE1} l'ensemble des points extrêmes supportés non-dominés et Y_{SE2} celui des points non-extrêmes supportés non-dominés.

La figure 1.1 récapitule les classes de points et leur dénomination.

1.2.2 Ensembles bornants

L'encadrement de l'ensemble des solutions d'un problème est une question importante en optimisation. Dans un problème mono-objectif, l'encadrement des solutions s'effectue grâce à une borne supérieure et une borne inférieure.

Définition 4. *(Bornes pour un problème mono-objectif) Soit x^* une solution optimale pour le problème P.*

— *Une borne supérieure sur $z(x^*)$ est une valeur $u \in \mathbb{R}$ telle que $z(x) \leq u$ pour tout $x \in \mathbb{R}^n$*
— *Une borne inférieure sur $z(x^*)$ est une valeur $l \in \mathbb{R}$ telle que $l \leq z(x)$ pour tout $x \in \mathbb{R}^n$*

A partir de deux objectifs, l'utilisation de ces bornes fonctionne beaucoup moins bien. En gardant la même formulation, on obtient des bornes très mauvaises et non exploitables en pratique.

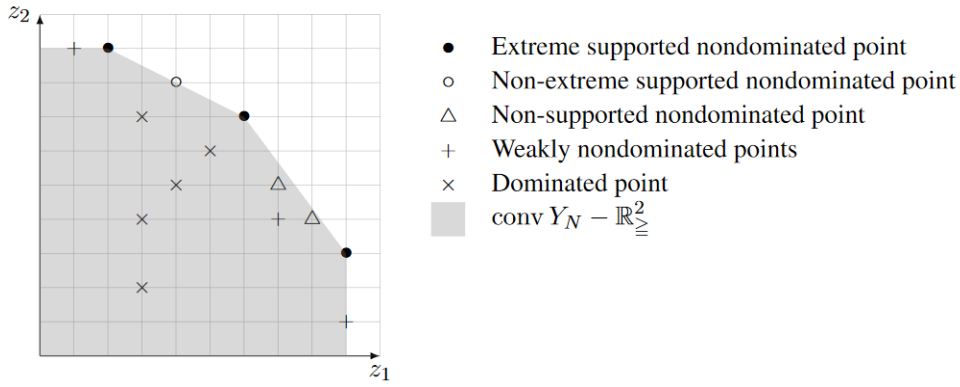


FIGURE 1.1 – Illustration des classes de points présentés - issu de [**?**], p.27

Définition 5. (BORNES POUR UN PROBLÈME MULTI-OBJECTIFS)
- *Une borne supérieure à Y_N est un point $u = (u_1, ..., u_p) \in \mathbb{R}^p$ tel que pour tout $y \in Y_N$ et pour tout $k \in 1, ..., p$, $y_k \leq u_k$.*
- *Une borne inférieure à Y_N est un point $l = (l_1, ..., l_p) \in \mathbb{R}^p$ tel que pour tout $y \in Y_N$ et pour tout $k \in 1, ..., p$, $l_k \leq y_k$.*

Une borne supérieure doit ainsi dominer tous les points non dominés. Les deux meilleurs points qui répondent à cette dernière définition sont les points idéal et nadir.

Définition 6. (POINTS IDÉAL ET NADIR)
Pour un problème en maximisation on peut définir :
- *Le point idéal est un point y^I tel que $y^I_k = max\{y_k : y \in Y_N, k = 1, ..., p\}$.*
- *Le point nadir est un point y^N tel que $y^N_k = min\{y_k : y \in Y_N, k = 1, ..., p\}$.*

Ces points sont ainsi par définition très éloignés des points non dominés ce qui fait d'eux de très mauvaises bornes.

Pour pallier à ce problème, la notion d'ensembles bornants a été introduite en 1981 par Villarreal *et al.*[9]. Cette fois, on ne requière plus une domination complète de Y_N par un seul point mais une domination point par point.

Définition 7. (ENSEMBLES BORNANTS - *(Villarreal et Karwan 1981)) Un ensemble bornant supérieur U pour un problème (MOILP) P est un ensemble de points qui respectent les deux conditions suivantes :*
- *Chaque point de U est soit un point non dominé ou domine au moins un point non dominé du problème P.*
- *Chaque point non dominé de P est dominé par au moins un élément de U ou est lui-même un élément de U.*

Un ensemble bornant inférieur L pour un problème (MOILP) P est un ensemble de points qui respectent la condition suivante :
- *Chaque élément de L est soit un point non dominé soit est dominé au moins par un point non dominé de P.*

Ehrgott et Gandibleux[3] donnent en 2001 une autre définition des ensembles bornants pour les solutions d'un problème multi-objectifs.

Définition 8. (ENSEMBLES BORNANTS - *(Ehrgott et Gandibleux, 2001))*
Soit $\bar{X}$ un sous ensemble de X_E et $\bar{Y} = Z(\bar{X})$. Un ensemble bornant supérieur pour Y_N est un ensemble U tel que :
- *Pour tout point y de $\bar{Y}$, il existe un $u \in U$ qui le domine.*
- *Il n'existe pas de couple $y \in \bar{Y}, u \in U$, tel que y domine u.*

Un ensemble bornant inférieur pour Y_N est un ensemble L tel que :
- *Pour tout point y de $\bar{Y}$, il existe un $l \in L$ qui est dominé par y.*
- *Il n'existe pas de couple $y \in \bar{Y}, l \in L$, tel que l domine y.*

Définition 9. (ENSEMBLES BORNANTS - *(Ehrgott et Gandibleux, 2007))*
Soit $\bar{Y} \subset Y_N$.
Un ensemble bornant supérieur U pour $\bar{Y}$ *est un ensemble* $\mathbb{R}^p_{\leqq}$*-fermé et* $\mathbb{R}^p_{\leqq}$*-borné* $U \subset \mathbb{R}^p$ *tel que* $\bar{Y} \subset U - \mathbb{R}^p_{\leqq}$ *et* $U \subset (U - \mathbb{R}^p_{\leqq})_N$.
Un ensemble bornant inférieur L pour $\bar{Y}$ *est un ensemble* $\mathbb{R}^p_{\leqq}$*-fermé et* $\mathbb{R}^p_{\leqq}$*-borné* $L \subset \mathbb{R}^p_{\leqq}$ *tel que* $\bar{Y} \in cl[(L - \mathbb{R}^p_{\leqq})^c]$ *et* $L \subset (L - \mathbb{R}^p_{\leqq})_N$.

On utilisera la définition de 2007, ce qui permet de considérer Y_N comme un possible ensemble bornant supérieur.

Définissons à présent la notion de dominance entre deux ensembles bornants.

Définition 10. *(Dominance pour les ensembles bornants supérieurs - (Ehrgott et Gandibleux 2007))*
Soit deux ensembles bornants supérieurs U^1 *et* U^2 *pour* $\bar{Y}$ *:* U^1 *domine* U^2 *si* $U^1 \subset U^2 - R^p_{\geq}$ *et* $U^1 - R^p_{\geq} \neq U^2 - R^p_{\geq}$.

On dit ainsi qu'un ensemble bornant supérieur domine un autre s'il est "au dessus" de lui. En pratique, il est rare d'observer un tel cas de figure. On observe plus fréquemment des recouvrements partiels. Par exemple, U^1 est au dessus de U^2 par endroits, et U^2 au dessus à d'autres endroits.

Ehrgott et Gandibleux[4] ont proposé en 2007 une "fusion" de plusieurs ensembles bornants afin d'en obtenir un nouveau, meilleur que tous les autres.

Proposition 1. *((Ehrgott et Gandibleux 2007))*
Soit deux ensembles bornants supérieurs U^1 *et* U^2 *pour* $\bar{Y}$ *tels que* $U^1 - R^p_{\geq} \neq U^2 - R^p_{\geq}$ *alors* $U^* = [(U^1 - R^p_{\geq}) \cap (U^2 - R^p_{\geq})]_N$ *est un ensemble bornant supérieur dominant* U^1 *et* U^2.

1.3 Méthode de résolution

Parallèlement aux méthodes de résolution de problèmes mono-objectif en variables entières , il existe plusieurs méthodes différentes, plus ou moins efficaces pour résoudre de manière exacte un problème multi-objectifs.

De même que pour le cas mono-objectifs, il existe un algorithme du simplexe multi-objectif pour les problèmes en variables continues. Pour les problèmes en variables entières cependant, peu de méthodes rapides existent. Nous allons ici en présenter deux, qui seront au coeur de ce TER, le branch-and-bound et l'ϵ-contrainte.

1.3.1 ϵ-contrainte

La méthode de l'ϵ-contrainte permet de trouver l'ensemble de solutions non-dominées, c'est à dire de résoudre le problème multi-objectifs. L'idée principale est de conserver une unique fonction objectif et transformer la seconde en une contrainte. Le problème obtenu - appelé ϵ-problème - est présenté ci-dessous :

$$\begin{bmatrix} max\ \mathrm{z}_1(\mathrm{x}) \\ \text{s.t.} \quad \mathrm{z}_2(\mathrm{x}) > \epsilon \\ x \in X \end{bmatrix}$$

En affectant différentes valeurs à ϵ, la méthode résout itérativement le problème présenté ci-dessus. La première itération consiste à trouver la solution optimale en ne considérant qu'une des deux fonctions objectif initiale. Lors de l'itération suivante, on fixe ϵ de façon à ce que la nouvelle ϵ-contrainte rende la dernière solution trouvée non admissible. L'exécution s'arrête lorsque plus aucune solution n'est trouvée.

On a présenté ici l'ϵ-contrainte pour un problème à deux fonctions objectif, mais cette méthode s'étend à $p > 2$ objectifs en gardant une fonction objectif et en transformant les autres en contraintes. De ce fait, on crée $\epsilon_1, ..., \epsilon_{p-1}$ contraintes. On va ensuite effectuer le même principe que précédemment en fixant tous les ϵ et en n'en faisant varier un pour trouver toutes les solutions.

Dans le cas bi-objectif (i.e deux fonctions objectifs), cette méthode se trouve être l'une des plus performantes.

1.3.2 Branch-and-bound

Parmi toutes les méthodes différentes de résolution de problèmes multi-objectifs, les branch-and-bound font partis de ceux étant prometteurs pour avoir de bons résultats. La recherche d'ensembles bornant efficace est effectuée pour pouvoir améliorer au mieux ces algorithmes pour ainsi obtenir des temps rivalisant avec les meilleurs méthodes.

a) Explication du cas mono-objectif

Tout d'abord rappelons le principe d'un branch-and-bound mono-objectif. Premièrement, un ordre est appliqué sur les variables, généralement issu de la comparaison des variables via une certaine fonction d'utilité. Ensuite, à partir d'une solution initiale, on fixe la première variable (selon l'ordre instauré) à 1 et on étudie le sous-problème associé. Si la borne supérieure de ce sous-problème est inférieure à notre meilleure solution actuelle, alors cela ne sert à rien d'étudier ce sous problème et celui-ci est sondé par dominance. On continue l'assignement progressif des variables, à 1 et à 0, jusqu'à ce que tous les sous-problème à étudier soient sondés.

Il y a 3 moyens de sonder un sous-problème :

1. par dominance : si une borne supérieure est inférieure à la valeur de la solution primale ;
2. par infaisabilité : si le sous-problème n'est simplement pas faisable ;
3. par optimalité : si la valeur de la borne supérieure est égale la valeur de la solution primale.

À présent, explorons comment le branch-and-bound peut être appliqué à un contexte multi-objectifs.

b) Application au multi-objectifs

En passant du mono au multi-objectifs, les bornes – inférieure et supérieure – qui étaient alors chacune une solution, deviennent un ensemble de solutions.

1. L'ensemble bornant inférieur est une liste de points correspondants aux solutions admissibles non dominées de notre problème. Cet ensemble fait office de borne inférieure car si une solution est dominée par un point de cet ensemble, alors cette solution ne peut être non dominée. Cet ensemble sera aussi défini par le terme borne primale.
2. L'ensemble bornant supérieur est un ensemble de segments. Il est défini de telle manière qu'aucune solution admissible ne peut se trouver au dessus d'un des segments. Cet ensemble sera aussi nommé par le terme borne duale.

L'algorithme débute par calculer une première borne primale (ensemble bornant inférieur). Ce calcul est effectué par un algorithme de dichotomie.

Les solutions non-supportées non-dominées ne peuvent se trouver que dans les triangles formés par les points de l'ensemble bornant inférieur et leurs points nadir respectifs. Ces triangles sont illustrés par les zones grises dans le graphique ci-dessous :

Puisque les solutions recherchées sont essentiellement les solutions non-dominées, les recherches du branch-and-bound peuvent se restreindre à ces triangles. Plus précisément, si un sous-problème ne peut pas avoir de solution se trouvant dans ces triangles, alors il est inutile de l'étudier (sondage par dominance).

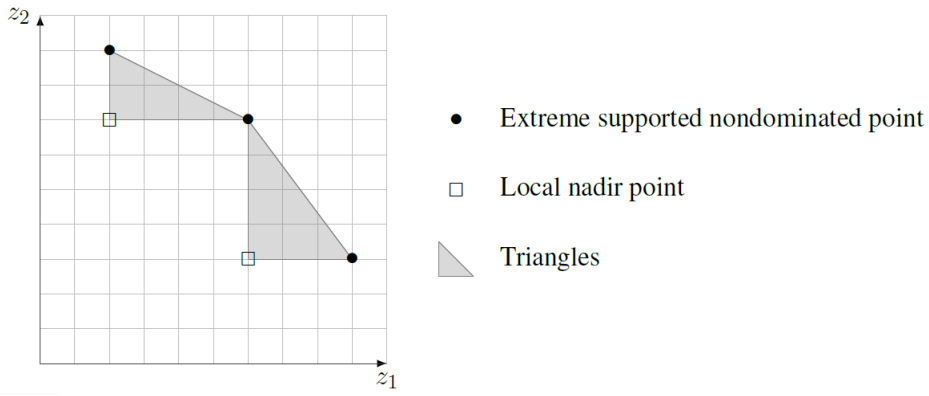


FIGURE 1.2 – Illustration des triangles comportant les points non-dominés - issu de [**?**], p.44

Comme pour le branch-and-bound mono-objectif, les variables ont intérêt à être triées selon un ordre prédéfini. Cet ordre a pour fonction de d'abord brancher sur les variables importantes, afin de pouvoir plus rapidement sonder et ainsi de restreindre un maximum la profondeur de l'arbre. Un ordre efficace pour du multi-objectifs est plus difficile à déterminer que pour du mono-objectif et dépend de la structure du problème. Nous situant dans un problème à multi-objectifs, les utilités des variables ne peuvent plus être calculées trivialement comme dans le cas mono-objectif. Une variable intéressante sur une fonction objectif ne le sera pas systématiquement sur les autres. Comme évoqué par Cerqueus [], plusieurs ordres peuvent considérés. On pense par exemple à faire la moyenne des utilités pour chaque fonction, ou bien sélectionner la meilleure utilité. Les résultats présentés dans l'article ne sont pas prometteur, nous préférons donc nous focaliser sur d'autres paramétrages.

La méthode de résolution/branchage reste la même. Les variables sont assignées les unes après les autres, et chaque sous-problème engendré par un nouvel assignement est étudié.

L'étude du sous-problème commence par le calcul de son ensemble bornant supérieur. Ce calcul varie en fonction des choix d'implémentation. Il peut être le résultat d'une relaxation linéaire sur le sous problème, ou le résultat d'une dichotomie. Si la relaxation linéaire est choisie, l'ensemble bornant supérieure sera plus lâche que l'enveloppe convexe produite par la dichotomie mais son calcul sera moins coûteux, une solution pourrait être d'utiliser aussi la relaxation convexe dans la dichotomie.

Une fois l'ensemble bornant supérieur du sous-problème construit, il faut déterminer si le sous-problème peut-être sondé. De même que pour le mono-objectif, il y a trois moyens de sonder un sous-problème :

1. par dominance : si l'intersection entre l'ensemble bornant supérieur du sous-problème et les triangles est vide. Ce cas de figure implique qu'en aucun cas une solution pour le sous problème pourra être une solution non-dominée pour le problème principal ;
2. par infaisabilité : si le sous-problème n'admet aucune solution admissible ;
3. par optimalité : ce cas est très rare, lorsqu'on est sûr que l'ensemble bornant inférieur ne peut plus être amélioré et qu'il est ainsi optimal. En pratique, ce cas arrive uniquement lorsque l'ensemble bornant supérieur du sous-problème est réduit à un unique point admissible.

Une fois qu'un sous-problème est sondé, on fixe la variable associée à ce sous-problème à 0 et on applique le même schéma. Le parcours de l'arbre du sous-problème se fait d'abord en profondeur.

On poursuit ainsi les itérations du branch-and-bound, jusqu'à ce qu'il n'y ait plus de sous-problèmes non-sondés.

A chaque fois que, dans un sous-problème, on trouve un point non-dominé et non présent dans notre ensemble bornant inférieur du problème principal, on l'ajoute à cet ensemble. Ainsi, lorsque l'exécution du branch-and-bound se termine, cet ensemble bornant inférieur contient tous les points non-dominés supportés et non supportés du problème principal.

c) Algorithme de dichotomie

La méthode dichotomique dans le contexte d'une résolution multi-objectifs, est un algorithme déterminant les points supportés extrêmes du problème considéré. Cette méthode se base sur la scalarisation de sommes pondérées P_λ présentée précédemment.

La première étape consiste à trouver les deux solutions optimales selon l'ordre lexicographique (x_1, x_2). x_1 (resp. x_2) est obtenue en résolvant le problème mono-objectif associé à la somme pondérée (1,ϵ) (resp. (ϵ,1)), avec $\epsilon > 0$. x_1 (resp. x_2) est donc la meilleure solution selon la première (resp. seconde) fonction objectif. Notons que ϵ remplace 0 de telle manière que dans le cas où il y aurait plus d'une meilleure solution selon un des objectif, la solution retenue ai, parmi les autres solutions candidates, la meilleure valeur pour le second objectif. ϵ prend généralement une valeur très proche de zéro.

La méthode itérative fonctionne comme suit. La recherche s'effectue par rapport à une paire de solutions extrêmes non dominées adjacentes. Supposons travailler sur la paire (x_1, x_2). Est alors calculé le vecteur λ orthogonal au segment (x_1, x_2). Le problème est résolu selon la somme pondéré associée au vecteur λ. Si une solution (x_3) améliorante est trouvée, alors les deux nouvelles paires (x_1, x_3) et (x_3, x_2) sont ajoutées à la liste des paires à étudier. Et la recherche dichotomique est récursivement appelée sur ces deux nouvelles paires. Si aucun nouveau point améliorant n'est trouvé, la recherche s'arrête de chercher sur la paire et passe à l'étude de la paire suivante qu'il reste à étudier.

Le vecteur λ correspond au vecteur $(y_2^l - y_2^r,\ y_1^r - y_1^l)$ pour la paire (x^l, x^r).

Cette méthode de permet pas de trouver les solutions supportées non-extrêmes. Elle fournit simplement une partie de l'enveloppe convexe des solutions du problème.

d) Problèmes et améliorations

La principale limitation de la méthode dichotomique est d'être restreinte aux problèmes bi-objectifs. En effet, le concept de point nadir se complexifie beaucoup à partir de trois objectifs, nécessitant trois points au lieu d'un seul. De plus, un espace en trois dimensions ou plus n'admet aucun ordre complet, ce qui rend impossible l'instauration d'une liste de points consécutifs.

Un autre problème est le nombre de fois où l'on doit calculer les ensembles bornants (chaque sous-problèmes). Pour pallier à ce problème, on peut utiliser quelques astuces d'implémentation. Tout d'abord, lors du calcul de l'ensemble bornant supérieur : la première idée aurait été d'appliquer la dichotomie entièrement pour avoir tout l'ensemble bornant supérieur et après cela vérifier l'appartenance des point nadir. L'idée ici est plutôt de vérifier l'appartenance de chaque point nadir pendant la création de l'ensemble bornant. Ainsi, si celui-ci au cours des premières itérations – même avec une forme très approximative – contient au moins un point nadir, alors on arrête de le construire et on continue l'étude.

De plus, lors du test des points nadir, si un point nadir n'appartient pas à l'ensemble bornant supérieur d'un sous-problème, alors ça ne sert à rien de continuer à chercher son appartenance dans les sous-problèmes fils. On peut donc alors restreindre les tests d'appartenance et ainsi limiter les différents calculs et la création d'un ensemble bornant supérieur très précis à chaque itération.

1.4 Le problème du sac à dos

1.4.1 Définition

Le problème du sac à dos unidimensionnel mono-objectif est un problème d'optimisation combinatoire consistant à sélectionner des objets sans dépasser une certaine capacité. Chaque objet possède ainsi un profit et un poids. L'objectif consiste à maximiser la somme des profits des objets sélectionnés sans pour autant que la somme des poids de ces objets dépasse une capacité donnée.

La généralisation de ce problème à un cadre multi-objectifs est la suivante. On ne considère plus un seul profit par objet mais p profits différents, un pour chaque objectif.

Plus formellement, le problème du sac à dos mono-dimensionnel multi-objectifs s'écrit comme suit :

$$\left[\begin{array}{lll} max\ \mathrm{z(x)} = \sum_{i=1}^{n} c_i x_i & & \\ \mathrm{s/c} & \sum_{i=1}^{n} p_i x_i \leq L & \\ & x_i \in \{0,1\} & i = 1,...,n \end{array}\right]$$

1.4.2 Spécificité du sac à dos multi-dimensionnels

Dans sa forme la plus générale le problème du sac à dos possède p objectifs, n variables et m contraintes. Ainsi le sac-à-dos multi-dimensionnels (avec au minimum 2 contraintes) est une forme plus générale, mais aussi bien plus complexe du problème du sac-à-dos. Beaucoup d'études ont été menées pour simplifier ce problème. C'est pour cette raison que nous allons rester sur le problème du sac-à-dos mono-dimensionnel, pour n'étudier que la différence entre les algorithmes de résolution et non la complexité des problèmes.

1.5 Direction et choix pris pour la réalisation du TER

Ce sujet nous permettant de nous intéresser aux ensembles bornants, et plus particulièrement à leurs applications dans les algorithmes de branch-and-bound, des choix ont dût être fait pour ne pas se perdre dans notre travail et garder un fil rouge cohérent. On va donc présenter ici les différentes décisions que nous avons prises.

1.5.1 Le type du branch-and-bound

Sans rentrer dans les détails algorithmiques, il existe au moins deux types de méthodes pour le branch-and-bound multi-objectif s :

— Un branch-and-bound général : cela consiste à faire exactement ce qui a été décrit précédemment sur le problème en entier et donc d'étudier tous les triangles de la borne primale constamment ;

— Un branch-and-bound spécifique à chaque triangle : cela consiste en un branch-and-bound, mais centré uniquement sur un triangle, et constitant à appliquer la méthode pour chacun d'entre eux.

Ces deux algorithmes nous ont d'abord parues équivalents, cependant l'algorithme de branch-and-bound spécifique aux triangles à déjà pu être étudié par Gauthier Soleilhac[10]. C'est pour cette raison que nous avons décider d'étudier un branch-and-bound global.

Une différence notable de ces deux méthodes est le prétraitement. En effet, le prétraitement pour un branch-and-bound local à chaque triangle sera bien plus précis puisque les prétraitements seront propres à chaque triangle, ainsi chacun sera différent. Contrairement au branch-and-bound local qui n'aura qu'un prétraitement général et donc bien moins précis que ne pourra l'être le branch-and-bound local.

1.5.2 Le problème

Comme dit dans l'introduction des notations et notions, nous avons choisi de nous focaliser sur le problème du sac à dos mono-dimensionnel bi-objectifs.

Le choix du bi-objectifs a été fait pour rester en accord avec notre problématique. Premièrement, un branch-and-bound sur au minimum trois objectifs est beaucoup plus dur à mettre en place, à cause du fait qu'il n'existe aucun ordre sur les solutions. Même s'il n'existe pas d'ordre réel dans un espace à deux dimensions, en triant les solutions non-dominées par ordre croissant suivant le premier objectif, on sait aussi que celle-ci seront triées par ordre décroissant par rapport au second objectif. De ce fait, la notion de solutions consécutives existe et permet d'effectuer l'algorithme de dichotomie présenter plus tôt. En dimension trois ou plus, ce principe ne tient plus et d'autres techniques doivent être utilisées, comme vu dans le papier d'Anthony Przybylski[7].

Le choix d'un problème de sac-à-dos mono-dimensionnel a lui été fait par soucis de simplicité. En effet, le sujet ne consiste pas en une étude du problème du sac-à-dos mais bien de l'algorithme de branch-and-bound. Il nous a donc semblé inutile de compliquer le sujet avec un problème aussi dur et coûteux que le sac-à-dos multi-dimensionnel.

Pour finir, le problème du sac-à-dos en lui-même a été choisi, étant le problème le plus étudié de l'optimisation de programme linaire en variables binaire. C'est un problème simple, connu de tous, avec une grande gamme d'instances, et une relaxation linéaire (sauf cas pathologiques) proche de la solution entière. Toutes ces raisons font de ce problème le plus utilisé pour décrire des algorithmes étudiant des programmes linéaires en variables binaires.

Pour résumer, on a donc choisi de comparer tous nos algorithmes sur le 2O-UKP (2 Objectives Unidimensional Knapsack Problem). C'est cette notation qui sera gardée pour la suite de ce papier.

Chapitre 2

Évolution de l'algorithme

2.1 Premières idées et algorithme

Sont présentées ci-après les premières idées de l'algorithme dans sa première version. Il est grandement inspiré de la thèse d'Audrey Cerqueus[1]. Cette première version fait office de témoin pour l'étude des autres algorithmes. il va nous servir de témoin pour le comparer à nos différents algorithmes qui ont évolués au cours des différentes améliorations. L'algorithme repose sur le calcul d'une enveloppe convexe.

2.1.1 Fonctionnement du code

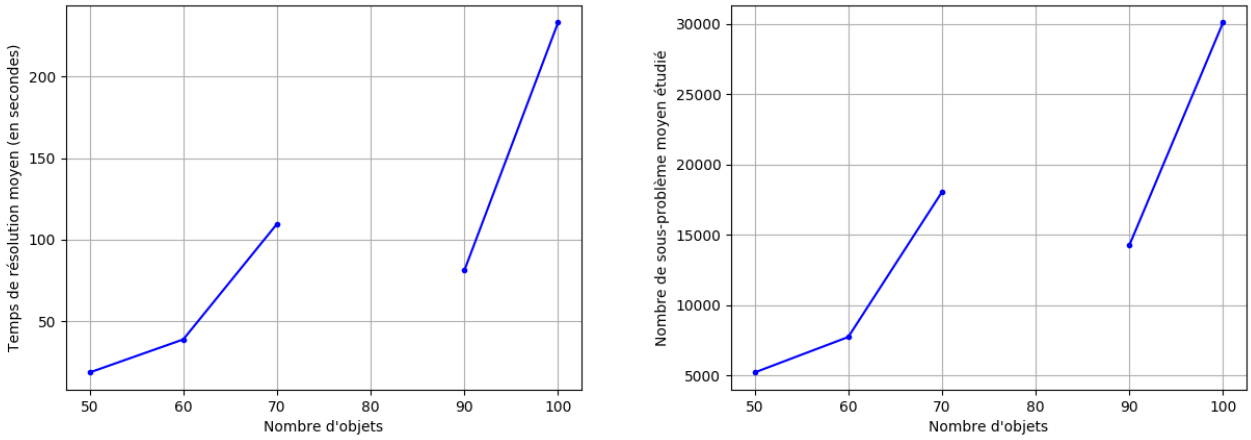
En s'inspirant de la description du fonctionnement du branch-and-bound, nous obtenons une première version de l'algorithme :

- on calcule d'abord l'ensemble bornant inférieur (LB) qui correspond aux points extrêmes de l'enveloppe convexe de l'ensemble des solutions ;
  - pour calculer l'enveloppe convexe on va effectuer l'algorithme de dichotomie en résolvant le problème du sac-à-dos mono-objectif associé avec JuMP/ GLPK, un solver disponible en Julia permettant de résoudre des problèmes d'optimisation ;
- on fixe ensuite la première variable non fixée à 1, on étudie le sous-problème associé en calculant son ensemble bornant supérieur avec la même méthode en retenant cette fois non pas les sommets mais les arrêtes de l'ensemble de solution, puis on vérifie que ce sous-problème est possiblement améliorant ;
- si de nouvelle solution apparaissent grâce à ce sous-problème, elle sont ajoutées à LB ;
- si au moins un des points nadir de LB appartient à l'ensemble bornant supérieur du sous-problème, alors on fixe la variable suivante à 1 et on recommence ;
- sinon, on sonde ce sous-problème et on fixe cette fois la variable à 0 et on recommence ;
- le programme s'arrête si le problème principal est sondé, alors on renvoie l'ensemble des solutions non-dominées du problème, LB ;

Cette première version nous a permis d'effectuer une première ébauche du code et d'ainsi mieux comprendre son fonctionnement ainsi que ce qui pouvait causer des ralentissements dans l'algorithme.

2.1.2 Résultats

L'algorithme est exécuté pour les instances aléatoires générées précédemment.



(a) (b)

FIGURE 2.1 – Temps de résolution en secondes – à gauche – et moyenne du nombre de sous-problèmes – à droite – pour des instances de 50 à 100 variables, avec l'algorithme basé sur le calcul de l'enveloppe convexe.

Comme présenté précédemment, on décide de tester l'algorithme sur des instances de 50 à 150 variables. On fournit donc cinq instances par pas de 10 variables. Ainsi, on étudie cinq instances à 50 variables, cinq autres à 60, et ainsi de suite jusqu'à 150. Cependant, connaissant les temps moyen d'exécution, certaines instances ne seront pas résolues en un temps acceptable. C'est pourquoi on impose un temps limite de 10 minutes par nombre de variables.

Dans le cas de cet algorithme, on voit que seules les instances jusqu'à 100 variables sont résolues durant le temps imparti. On note également qu'aucune donnée n'est disponible pour les instances à 80 variables. Cela ne signifie pas nécessairement que l'algorithme est trop long sur toutes les instances, mais qu'il nécessitait plus de 10 minutes pour résoudre la première instance.

Pour ce qui est du temps de résolution, on constate rapidement que l'algorithme n'est pas efficace. Pour un problème à 50 variables, l'exécution requiert en moyenne 19 secondes. A titre de comparaison, Audrey Cerqueus[1], dans sa thèse, présente des temps inférieurs à 10 secondes pour des problèmes à 105 variables. Il est cependant important de noter que, dans son cas, les problèmes sont corrélés contrairement aux problèmes étudiés ici, qui sont générés totalement aléatoirement.

On étudie également le nombre moyen d'itérations pour chaque nombre de variables. Tout comme les temps évoqués précédemment, ces données s'avèreront utiles dans la suite du travail, afin de comparer les différentes implémentations.

Du point de vue du nombre de sous-problèmes crées, on note que les temps pour chaque problème et le nombre d'itérations sont corrélés.

On en conclut donc que cette version est très lente et inutilisable en pratique. Elle présente tout de même l'intérêt de nous donner une première base et un moyen de comparaison, afin d'évaluer les améliorations que nous allons y apporter.

2.1.3 Problèmes rencontrés

Durant la programmation de cette algorithme, plusieurs problèmes sont apparus. Tout d'abord la nécessité que les éléments de la borne primale soit toujours triés, afin de constamment garder en mémoire l'ordre des points consécutifs. Naturellement, l'ajout d'élément devenait vite très coûteux.

La complexité de l'algorithme dépend énormément de la capacité du branch-and-bound à sonder par dominance. Un cas de figure qui complique ce sondage apparaît lorsque deux solutions non-dominées sont similaires. Il est parfois nécessaire d'avancer très profondément dans les branchements avant de trouver la seconde – qui était "camouflée" par la première.

De même, on retrouve le même problème pour prouver qu'une solution est effectivement non-dominée et qu'il n'existe donc pas d'autres solutions meilleures qu'elle.

Le dernier des problèmes rencontré lors de cette implémentation est le temps de calcul pris par la résolution du problème mono-objectif. On utilise actuellement le solveur GLPK, qui effectue un simplexe afin de résoudre chaque problème. C'est cette résolution qui consomme la majorité du temps de calcul de l'algorithme.

Pour pallier à ces différents problèmes, trois solutions ont été dans un premier temps misent en place, chacune pour améliorer un point différent.

2.2 Premières améliorations

2.2.1 Listes chaînées et relaxation linéaire

Les premières implémentations utilisaient des listes pour stocker les solutions de l'ensemble bornant primal. Notre utilisation de cette liste consiste essentiellement en parcours et insertion à n'importe quel endroit de la liste. Notre choix s'est donc ensuite porté sur un usage d'une liste chaînée afin de profiter d'un temps constant en insertion.

La relaxation linéaire apporte beaucoup de changement au comportement de l'algorithme. Au lieu de calculer l'ensemble bornant supérieur des sous-problèmes à l'aide d'une résolution exacte du problème mono-objectif lors de la méthode dichotomique, on utilise la méthode de la relaxation linéaire.

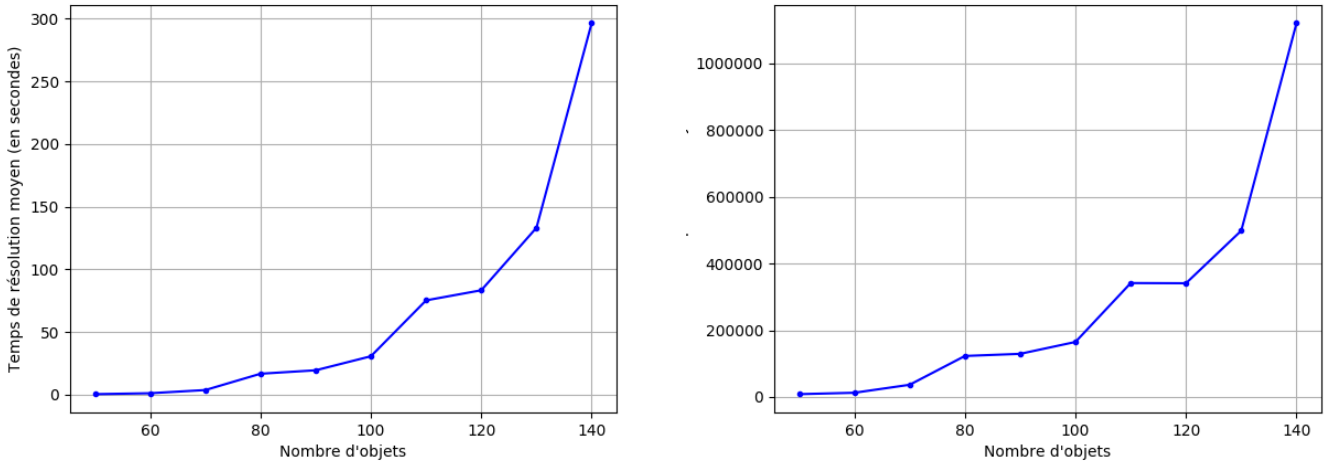
L'avantage de cette méthode est sa rapidité d'exécution – $\mathcal{O}(n \log n)$ comparé à du $\mathcal{O}(2^n)$ pour la résolution exacte. En contrepartie, l'ensemble bornant obtenu est plus lâche et le sondage des sous-problèmes est pénalisé.

Étudions les résultats pour valider l'intérêt de cette méthode.

2.2.2 Résultats

De la même façon que pour la première version de l'algorithme, nous exécutons cette nouvelle méthode sur les instances aléatoires. Les résultats sont présentés dans les graphes ci-dessous.

Au regarde de la figure 2.2, on constate que les temps sont considérablement réduits. En gardant le temps limite de 10 minutes pour chaque famille d'instance, on atteint les problèmes à 140 variables – contre ceux à 100 dans le cas du calcul de l'enveloppe convexe. Les problèmes à moins de 100 variables sont en moyenne résolus en moins de 50 secondes. On observe cependant une très grande marche entre les problèmes à 130 et 140 variables. Le temps nécessaire à la résolution – ainsi que le nombre de sous-problèmes – grandit exponentiellement. Pour ce qui est du nombre de sous-problèmes étudiés, on constate également une augmentation très forte. Dans le cas convexe, environ 30 000 sous-problèmes étaient étudiés pour les instances à 100 variables, contre presque 200 000 dans le cas de la relaxation linéaire.



(a) (b)

FIGURE 2.2 – Temps de résolution en secondes – à gauche – et moyenne du nombre de d'itérations – à droite – pour des instances de 50 à 100 variables, avec l'algorithme basé sur la relaxation linéaire.

On s'intéresse également aux nombres d'itérations moyens effectués par l'algorithme étudié. Ces données prennent tout leur sens lorsqu'on les met en parallèle avec les résultats obtenus par le calcul de l'enveloppe convexe. Pour ce dernier, on atteignait les 12 400 itérations pour les instances à 100 variables. Pour les instances à 100 variables, la relaxation linéaire effectue presque 200 000 d'itérations.

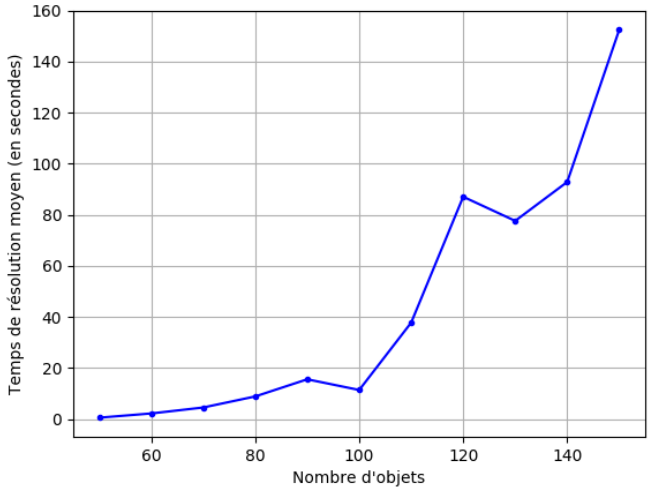
La relaxation linéaire provoque beaucoup plus de sous problèmes à étudier mais est pourtant beaucoup plus rapide que la méthode avec l'enveloppe convexe. Ce phénomène s'explique par le fait que la résolution exacte du problème du sac-à-dos par JuMP est très coûteuse. En passant par la relaxation linéaire, on gagne donc énormément de temps à ce niveau.

2.3 Combo

À la vue des résultats apportés par la relaxation linéaire, on peut s'apercevoir que calculer les ensembles bornants des sous-problèmes avec l'enveloppe convexe (JuMP/ GLPK) n'est plus une méthode viable. On peut pourtant voir que l'enveloppe convexe permet de réduire drastiquement le nombre de sous-problèmes étudiés par le branch-and-bound. L'idée a donc été de remplacer la résolution faite par JuMP et GLPK par un solver spécifique au problème du sac-à-dos mono-objectif mono-dimensionnel , qui permet lui aussi de résoudre, spécifiquement et de manière exacte, le problème du sac-à-dos. Nous avons donc utilisé le code de David Pisinger, Combo [2]. Ce code, publié en 1997 par D. Pisinger, S. Mathello, P. Toth, est toujours actuellement l'un des algorithmes les plus performants dans la résolution du problème du sac-à-dos mono-dimensionnel mono-objectif.

2.3.1 Résultats

Avec l'utilisation de la libraire Combo, les temps d'exécutions sont considérablement amoindris. Pour la première fois, un problème à 150 variables est résolu en moins de 10 minutes. Ces derniers nécessite en moyenne 150 secondes avant la résolution. A titre de comparaison, c'est environ le temps qu'il fallait pour traiter un problème à 130 variables avec la combinaison JuMP/GLPK. Les temps sont au minimum divisés par 5, voir par 10 dans certains cas. Cette amélioration est donc très performante et permet d'obtenir de bien meilleurs temps que ceux obtenus avec la relaxation linéaire.



(a)

FIGURE 2.3 – (a) Temps moyen de résolution (en secondes) pour des instances de 50 à 150 variables.

2.4 Une heuristique primale

2.4.1 Idée générale

On a constaté que le coût d'exécution du branch-and-bound provenait de la profondeur de l'arbre et par conséquent de l'incapacité de l'algorithme à sonder les sous-problèmes par dominance. L'algorithme peine à sonder par dominance lorsque les points nadirs sont mauvais, c'est à dire très éloignés de l'ensemble bornant inférieur. En effet, plus les triangles sont volumineux, plus l'ensemble bornant supérieur des sous-problèmes aura de chance de passer au dessus d'un point nadir et ainsi d'empêcher le sondage par dominance. Pour obtenir de meilleurs points nadirs, il est nécessaire d'enrichir l'ensemble bornant inférieur du problème principal. C'est l'objectif de l'heuristique primale que nous avons implémenté.

L'heuristique prend ainsi en entrée l'ensemble bornant inférieur initial, calculé par la méthode dichotomique (enveloppe convexe), et y ajoute de nouvelles solutions possiblement non dominées.

2.4.2 Algorithme

La méthode implémentée utilise une recherche locale en plus profonde descente. Sa particularité est qu'une fois un minimum local trouvé, la fonction d'évaluation est modifiée en conséquences et la recherche locale est appelée sur cette nouvelle fonction. La condition d'arrêt est vérifiée si la recherche locale n'a trouvée aucune nouvelle solution durant l'itération. Le prochain paragraphe apporte plus de détails sur la procédure exacte et le suivant présente un exemple didactique afin de clarifier le déroulé de l'algorithme.

Une recherche locale démarre depuis chaque solution non dominée connue (sommets des triangles) et se déplace de solution en solution. Chaque solution rencontrée est évaluée par une fonction d'évaluation. Comme pour une plus profonde descente classique, la recherche locale s'arrête dans un minimum local, c'est-à-dire qu'aucun voisin de la solution courante ne permet d'améliorer l'évaluation actuelle. Une fois que chacune des recherches locales est bloquée dans un minimum local, les nouvelles solutions non-dominées trouvées (solutions finales trouvées) sont ajoutées à l'ensemble bornant inférieur et une nouvelle itération de l'heuristique est exécutée. Lorsqu'une itération n'a trouvé aucun nouveau point, l'heuristique s'arrête.

Voici un exemple didactique. Supposons posséder un ensemble bornant inférieur contenant les solutions x_1, x_2, x_3. Une recherche locale est appelée sur chacune de ces trois solutions. Supposons maintenant que la recherche à partir de x_1 et x_3 ne découvrent aucune nouvelle solution non dominée, et que la recherche à partir de x_2 découvre la solution x_4. Alors une nouvelle itération est lancée cette fois à partir des quatres solutions x_1, x_2, x_3, x_4. Notons qu'il est possible que durant cette nouvelle itération x_1 et x_3 trouvent de nouvelles solutions. En effet, l'ajout de la solution x_4 modifie le calcul de la fonction d'évaluation, et rend ainsi chaque itération différente. Supposons maintenant que cette deuxième itération ne découvre aucune nouvelle solution. La recherche s'arrête alors et l'ensemble bornant inférieur x_1 jusqu'à x_4 est retourné.

Le graphe présenté ci-dessous illustre le comportement de la première itération de l'algorithme sur un problème à 70 variables :

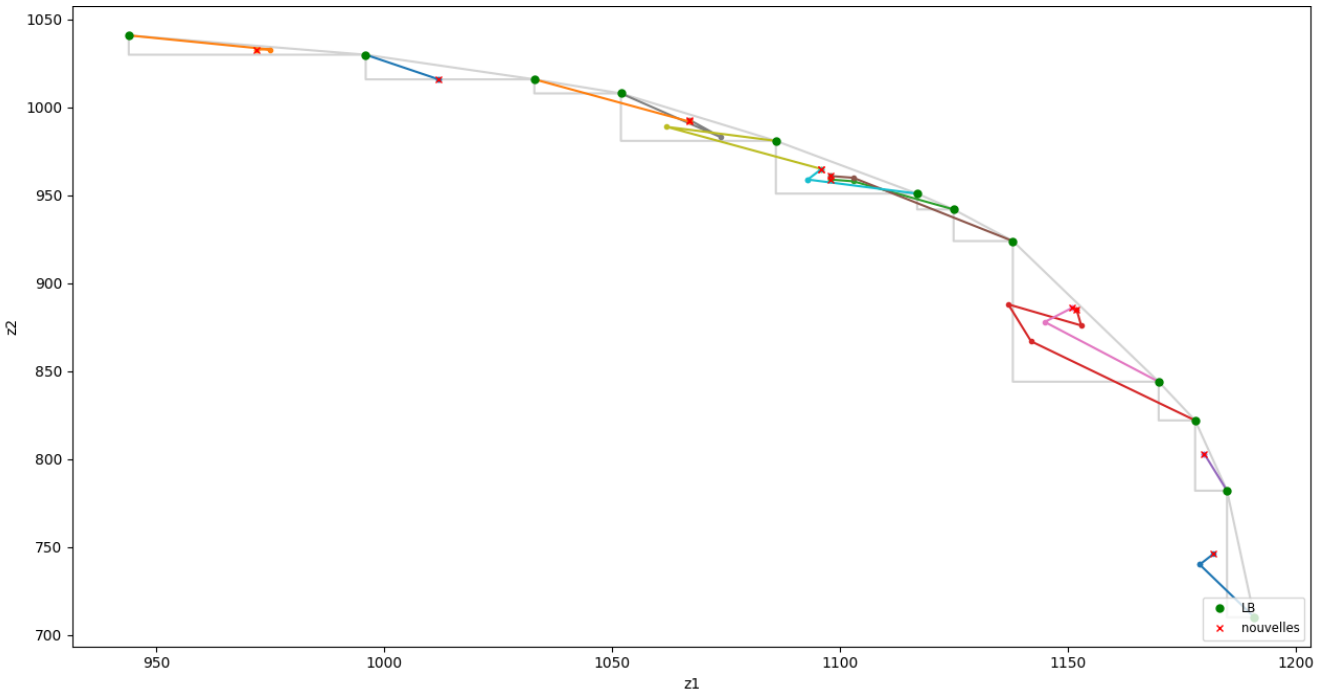


FIGURE 2.4 – Première itération de l'heuristique sur un problème à 70 variables

2.4.3 Fonction d'évaluation

La fonction d'évaluation est construite de manière à ce que le milieu de l'hypoténuse de chaque triangle soit un minimum local. Ainsi, chaque étape de la recherche locale terminant avec succès nous fournit une solution admissible un peu plus proche d'un des milieux des triangles.

Les voisinages utilisés par la recherche locale sont les kp-échanges 1-1, et kp-échanges-1-2. Un kp-échange-k-p consiste à retirer k objets du sac et à en ajouter p.

En plus de définir le milieu des hypoténuses comme des minimums locaux, la fonction d'évaluation attribue à chacun une profondeur différente. Cet aspect est justifié par le fait que les triangles ne sont pas tous aussi prometteurs. En effet, certains triangles, très fins, ont très peu de chance de contenir de nouvelles solutions. La fonction d'évaluation donne donc plus d'importance aux larges triangles. Comme on le voit sur la carte de chaleur présentée en figure 2.5, les larges triangles font l'objet de profondes vallées dans la fonction d'évaluation.

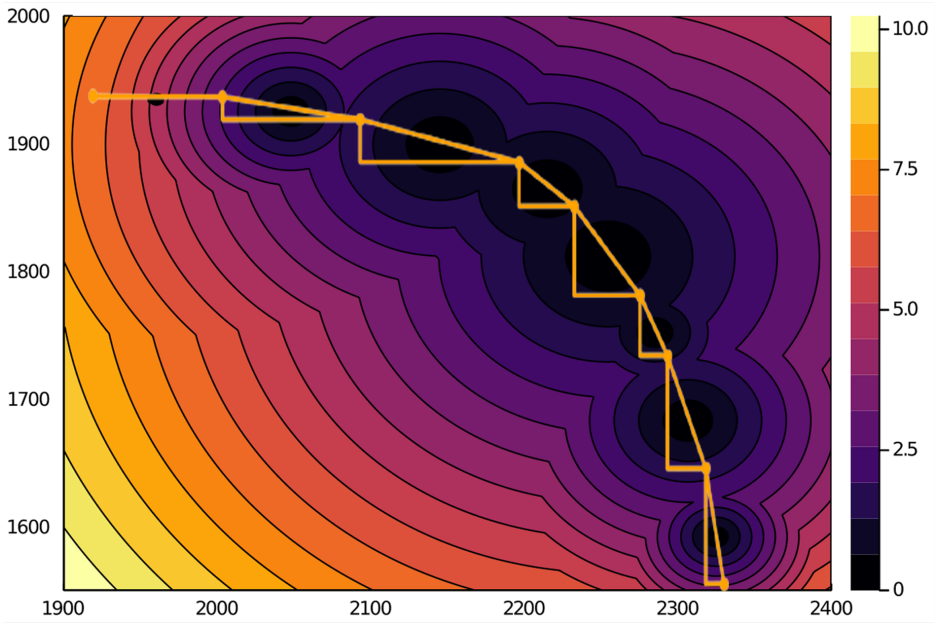


FIGURE 2.5 – Fonction d'évaluation

La fonction d'évaluation dépend de la solution et des triangles de l'ensemble bornant inférieur. Elle est calculée de la manière suivante :

$$y_t \qquad \text{(milieu de l'hypoténuse du triangle t)}$$

$$w_t = \frac{1}{min_{arrete \in t} longueur(arrete)} \qquad \text{(Poids du triangle t)}$$

$$S(y,t) = d(y_t, y)\, w_t \qquad \text{(score de la solution associé au triangle t)}$$

$$f(y,T) = min_{t \in T}\{S(y,t)\} \qquad \text{(Meilleur score de la solution parmi tous les triangles)}$$

Le poids du triangle w_t reflète inversement l'attractivité du triangle, évoquée précédemment. Plus le poids est élevé, moins le triangle est large et donc moins attractif. Le score de la solution associé au triangle t $S(y,t)$ est égale à la distance qui sépare la solution du milieu de l'hypoténuse du triangle, multipliée par le poids du triangle. Enfin la fonction d'évaluation correspond au minimum des scores de la solution par rapport à chacun des triangles de l'ensemble bornant inférieur actuel. La pondération par le poids du triangle de $S(y,t)$ permet que si une solution est proche d'un petit triangle et un peu plus loin d'un large triangle, le score de la solution par rapport au large triangle soit plus petite. Ce qui implique que la solution sera "attirée" par le minimum local correspondant au large triangle.

Est présenté en figure 2.6 un schéma illustrant très simplement l'évolution des minimums locaux de la fonction d'évaluation avant et après la première itération.

Les losanges violets correspondent aux solutions non dominées initiales, celles qui sont calculées par le méthode dichotomique. Le disque vert foncé est le milieu de l'hypoténuse du triangle formé par les solutions non dominées initiales. Il correspond donc au minimum local de la fonction d'évaluation pour la première itération. Au cours de cette itération, deux solutions sont trouvées, qui sont marquées par deux losanges rouges. Le triangle initial est alors décomposé en trois nouveaux triangles. Les cercles verts clairs correspondent aux milieux des hypoténuses de ces triangles. Ils représentent donc les minimums locaux de la nouvelles fonction d'évaluation, celle utilisée pour la seconde itération de l'algorithme. Le schéma s'arrête à la première itération, mais en réalité, la recherche locale continue tant qu'elle trouve une nouvelle solution durant l'itération.

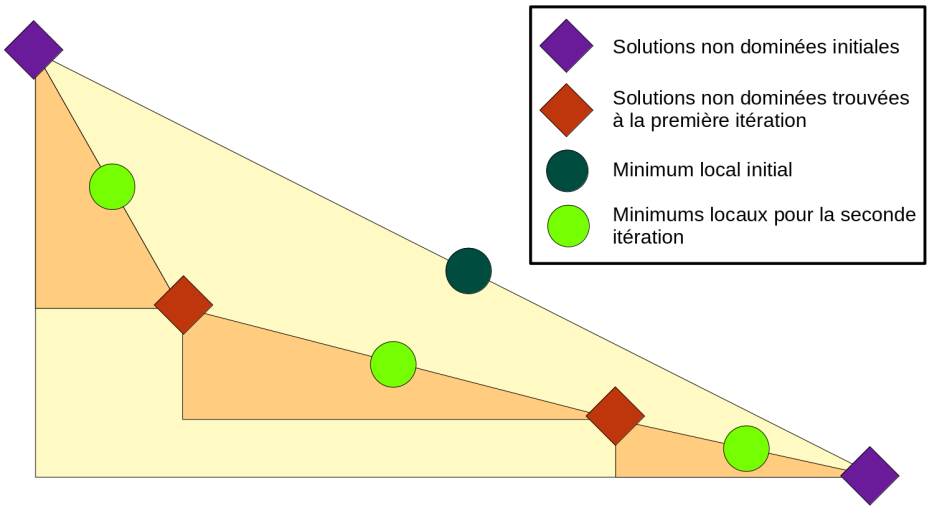


FIGURE 2.6 – Évolution des minimums locaux

Notons un point important qui est que les solutions rencontrées lors de chaque recherche locale ne sont pas toutes situées dans un triangle. En effet, la topologie de la fonction d'évaluation relâche la contrainte d'appartenance à un des triangles. En pratique on constate que peu de recherches sortent d'un triangle et n'y reviennent jamais. Tout dépend de la taille des triangles et du nombre de solutions non dominées qu'ils contiennent. Bien sûr, si une recherche locale aboutie sur une solution n'appartenant pas à un triangle, alors la solution n'est pas ajoutée à l'ensemble bornant inférieur. Des solutions en dehors des triangles n'apporteraient aucune nouvelle information pour le branch-and-bound.

Les solutions trouvées par l'heuristique ne sont pas forcément non dominées. En effet, même si la solution se trouve dans un triangle, il est possible qu'elle soit dominée par une solution encore inconnue, mieux placée dans le triangle. Cette particularité ne gêne pourtant pas le branch-and-bound, puisqu'à chaque nouvel ajout de solution (par le branch-and-bound), il supprime les solutions de l'ensemble bornant inférieur qui se révèlent être dominées.

2.4.4 Résultats

La comparaison des deux configurations – comme on peut le voir sur la figure 2.7 – met en exergue l'amélioration apportée par l'heuristique, en terme de temps de résolution. Bien que la différence soit à peine visible sur les plus petites instances (de 50 à 100 variables), le résultat est sans appel lorsque l'on dépasse les 110 variables. Par exemple, sans heuristique, un problème à 120 variables nécessite environ 200 secondes. En ajoutant l'heuristique, ce temps passe à 80 secondes.

2.5 Méthode de calcul améliorante pour la relaxation

Comme on a pu le voir, l'amélioration apporté par la librairie Combo de D. Pisinger améliore grandement le temps de calcul de l'algorithme. Il n'est plus comparable à la précédente version de la résolution à l'aide de l'enveloppe convexe. Et il a de plus dépassé l'algorithme de la relaxation linéaire. De ce fait, on a décidé d'étudier différentes hypothèses pour améliorer celle-ci, pour qu'elle puisse de nouveau rivaliser avec la librairie Combo et sa méthode utilisant l'enveloppe convexe.

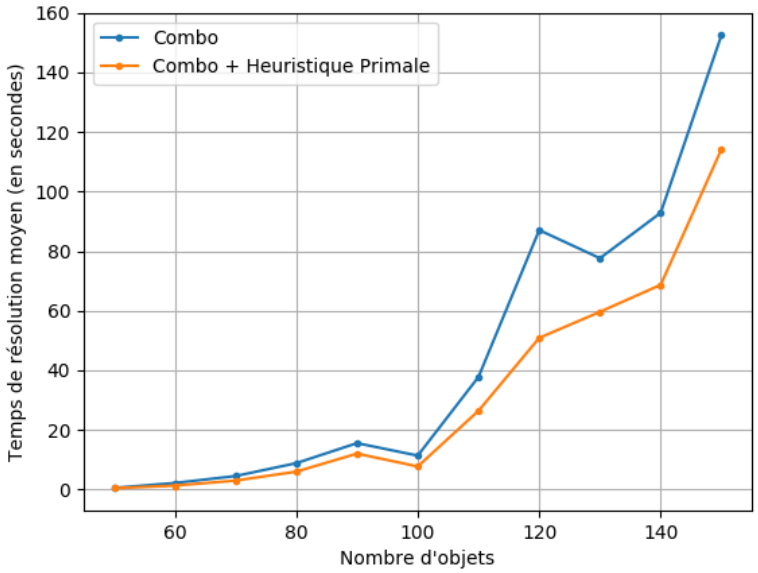


FIGURE 2.7 – (a) Moyenne de temps d'exécution de la relaxation linéaire sans heuristique contre moyenne des temps d'exécution de la relaxation linéaire avec heuristique

2.5.1 Explication de la méthode

L'idée principale est de réduire le plus possible le temps de calcule de la relaxation linéaire. Comme on a put le voir précédemment, le nombre de sous-problème étudié est imposant, optimiser encore plus cette méthode parait donc être une bonne idée.

En étudiant le calcul de la relaxation linéaire de plus près, on comprend que ce qui prend le plus de temps est le tri obligatoire des objets par rapport à leur utilité. Une fois cela fait, on parcourt toute la liste dans l'ordre de ce tri en rentrant au fur et à mesure les objets dans le sac. Malgré tout cette méthode est redondante, en effet l'ordre des utilités variant très peu, on parcourt la plupart du temps des listes très proches, ne faisant varier que peu la solution.

L'idée est donc de calculer pour un sous-problème, quels seront les scalaires λ critiques qui provoqueront un changement dans l'ordre des utilités des objets du sac à dos pour la fonction objectif :

$$\lambda z_1(x) + (1-\lambda) z_2(x)$$

Il ne restera donc plus qu'à trouver quelle solution correspond à chaque scalaire λ et pour un scalaire donné on pourra retrouver la solution de la relaxation linéaire.

- Pour chaque paire d'objet, on calcul la valeur de λ qui provoque l'échange de ces deux objets dans l'ordre de leurs utilités ;
- Si cette valeur n'appartient pas à $[0,1]$ alors cet échange ne peut pas s'effectuer ;
- on calcule d'abord la solution pour $\lambda = 1$
- Ensuite pour chacun des λ calculés précédemment, triés dans l'ordre décroissant, on effectue l'échange lié à ce λ et on calcule la nouvelle solution crée ;
- Chaque solution sera donc associée à un intervalle de validité pour la valeur de λ

2.5.2 Problèmes rencontrés

La réalisation de cet algorithme a entraînée beaucoup de problèmes dans le code. Le problème le plus important provoque un mauvais tris des solutions dans la liste de toutes les relaxations linéaires différentes. Ceci peut-être provoqué par un mauvais calcul des solution dans l'algorithme.

C'est à cause de plusieurs problèmes comme celui-ci que les expérimentations n'ont pas pus être réalisées.

2.6 Rappel des quatre algorithmes actuels et validation de l'algorithme le plus performant

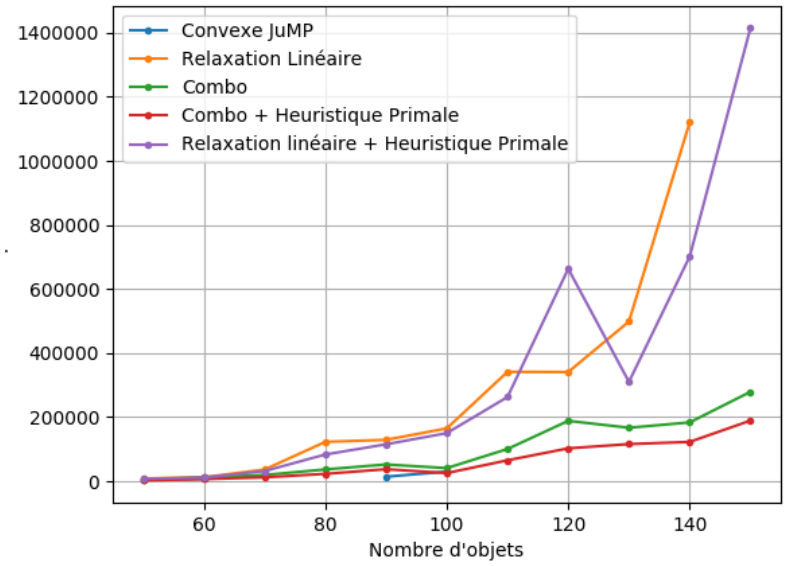


FIGURE 2.8 – Graphique du nombre d'itérations nécessaires à chaque configuration pour les instances étudiées.

Lorsque l'on compare les cinq configurations étudiées, on constate que l'amélioration apportée par Combo permet aux algorithmes qui l'utilisent de faire beaucoup moins d'itérations. C'est le cas des droites verte et rouge sur la figure 2.8. Comme expliqué précédemment, les relaxations linéaires vont effectuer beaucoup plus d'itérations. La mise en place de l'heuristique va presque systématiquement permettre de réduire ce nombre.

Cependant, il est important de rappeler qu'un grand nombre d'itérations n'est pas synonyme de lenteur lors de l'exécution. C'est pourquoi nous nous intéressons également à la comparaison des temps nécessaires à la résolution des problèmes selon les cinq configurations.

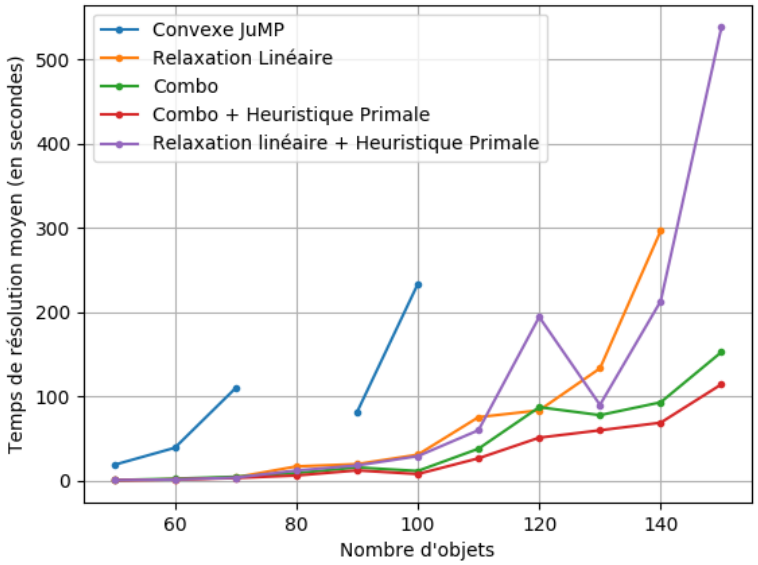


FIGURE 2.9 – Graphique des temps nécessaire à chaque configuration pour résoudre les instances étudiées.

En observant la figure 2.9, représentant le temps de calcul moyen de chaque algorithme en fonction du nombre d'objet, on peut remarquer le faite que certains algorithmes sont plus performant que d'autres.

Tout d'abord, si on étudie Combo et Combo plus l'heuristique primale, l'algorithme ayant le plus bas temps de résolution est Combo plus l'heuristique. Cela s'explique parce que, malgré le rajout de temps léger de la part de l'heuristique, celle-ci permet de sonder des sous-problèmes plus rapidement et de ne pas avoir à rajouter constamment des solutions dans l'ensemble bornant inférieur. Cette nette différence est cependant moins visible entre la relaxation linéaire avec et sans heuristique. Le temps de calcul étant déjà élevé, il avoisine pour les deux algorithme les 10 minutes pour 150 variables.

En comparant cette fois les algorithmes de relaxation linéaire et ceux utilisant Combo, ces derniers dominent pour toutes les instances la relaxation linéaire. De plus pour des problèmes à 150 variables, les méthodes utilisant Combo sont 6 fois plus rapide avec un temps moyen pour Combo avec l'heuristique avoisinant les 100 secondes.

Ainsi, entre les quatres derniers algorithmes, un se détache particulièrement des autres, Combo avec l'heuristique. Il permet d'obtenir des temps n'étant pas absurde (ne dépassant pas les 2 minutes). Cependant, un problème à 150 variables n'est pas représentatif des cas réels qui pourraientt demander à être résolu.

Chapitre 3

Implémentation du code général

L'objectif de ce code est donc d'appliquer le principe d'un Branch and Bound pour un problème Knapsack Bi-objectif. Le but va donc être de d'abord calculer une $lowerBound$ au problème principal, puis de séparer le problème en deux sous-problèmes, en fixant une variable à 1 puis à 0. Ensuite de regarder si le problème est utile pour améliorer notre $lowerBound$ si c'est le cas on recommence, sinon on s'arrête là.

3.1 Structures de donnée

a) Problem

```
struct Problem
    nbVar::Int
    nbObj::Int
    profits::Array{Float64, 2}
    weights::Array{Float64, 1}
    maxWeight::Float64
    isInteger::Bool
end
```

Une structure de donnée simple - appelée `Problem` - est utilisée pour garder en mémoire un problème Knapsack mono-dimensionnel, bi-objectif (2OUKP) spécifiquement. Le problème accepte des problèmes à coefficients entiers et réels.

On dispose ainsi de $nbVar$ - qui stocke le nombre de variables du problème, $nbObj$ qui retient le nombre d'objectifs (obligatoirement 2). On a également $profits$ - stockant tous les coefficients des fonctions objectifs. On accède donc aux coefficient de la variable $iterVar$ de la fonction objectif $iterObj$ grâce à $profits[iterObj, iterVar]$. Ensuite $weights$ correspond au vecteur des coefficients de la contrainte et $maxWeight$ est le poids maximal du sac à dos. $isInteger$ permet de savoir si le problème est entier ou non. Cela évite ainsi d'avoir à parcourir le problème pour avoir cette information.

On dispose également de plusieurs constructeurs. Un premier permet de créer un problème à 6 variables, déjà étudié et bien connu. On peut également créer un problème à partir d'un fichier texte, comportant toutes les données. Finalement, on peut aussi créer un problème en précisant tous les attributs décrits plus haut.

b) Solution

```
struct Solution
    x::Vector{Bool}
    y::Vector{Int}
    w::Int
    isInt::Bool
end
```

Une structure simple pour stocker une solution à ce problème a également été créee. Elle contient donc un vecteur x – stockant la valeur des variables, un vecteur y – stockant la valeur des différentes fonctions objectif et w – le poids total de la solution. On retrouve ici aussi un booléen $isInt$, qui permet de savoir si la solution est entière ou réelle. Ce paramètre est particulièrement utile lors du calcul de la relaxation linéaire.

c) Stockage de l'ensemble bornant inférieur

L'ensemble bornant inférieur, comme présenté précédemment, revient en une liste de solution admissible, non-dominée, qui évolue pour s'améliorer au fur et à mesure des itérations.

Ainsi pour stocker simplement la borne inférieur, on peut simplement utiliser un vecteur de $Solution$. Le problème est que l'on veut retenir ces points dans l'ordre (pour avoir accès aux différents points Nadir que créer ces solutions). Il existe donc 2 possibilités d'implémentations différentes, la première est de stocker tout les solutions dans l'ordre croissant par rapport au premier objectif, ainsi L'accès au points Nadir est très facile. La deuxième est d'utiliser un nouveau vecteur $consecutiveSet$ qui contiendra les paires de solutions consécutive, ainsi peu importe l'ordre dans lequel est trié notre $lowerBound$ puisque l'ordre sera conservé dans $consecutiveSet$.

Les différences entre ces deux implémentations seront expliqués dans la partie : Améliorations et optimisations possibles. Dans le code, le choix a été fait d'utiliser $consecutiveSet$

d) Stockage de l'ensemble bornant supérieur

Comme vu précédemment, l'ensemble bornant supérieur correspond à un ensemble d'arètes formant un polytope. Le choix pour représenter informatiquement l'ensemble bornant inférieur a été de le stocker comme un ensemble de contraintes (comme pour un problème d'optimisation). Ainsi si un point respecte toutes ces contraintes alors il appartient à cette ensemble bornant supérieur et inversement.

Le structure utilisée se représente comme ceci :

```
struct DualSet
    A::Array{Float64, 2}
    b::Array{Float64, 1}
end
```

La matrice A correspond donc aux coefficient des contraintes (similaire à un problème à deux variables puisqu'il n'y a que deux objectifs). Quand au vecteur b, il représente le membre de droite des contraintes.

e) Assignment

Comme nous utilisons un algorithme de type Branch and Bound, on effectue différents appels de fonction pour des sous-problèmes du probème principal. Ainsi, plutôt que de recréer à chaque appel un nouveau problème, on utilise une structure de donnée $Assignment$ qui s'occupe de stocker l'assignement (Les valeurs des différents variables fixées, le poids de celles-ci et leur profit) étudié.

```
mutable struct Assignment
    assign::Vector{Int}
    profit::Vector{Float64}
    weight::Float64
    assignEndIndex::Int
end
```

Le vecteur $assign$ est donc un vecteur de la même taille que le nombre de variables. Chaque variable fixée est mise à 0 ou à 1 - en accord avec la fixation - dans ce vecteur. Si par contre une variable est libre, elle est mise à -1. L'entier $assignEndIndex$, quant à lui, stocke l'indice de la dernière variable fixée par l'assignment. Les variables sont fixées à une valeur, dans l'ordre de gauche à droite de ce vecteur (d'abord la variable 1, puis 2 etc...). On sait ainsi que toutes les variables jusqu'à $assignEndIndex$ sont fixées.

3.2 Les grandes lignes de l'algorithme

L'algorithme implémenté prend en entrée un problème et retourne l'ensemble minimal de ses solutions non dominées.

D'abord, un premier ensemble bornant inférieur est calculé par une dichotomie. Ensuite, cet ensemble de solutions extrêmes est fourni à la fonction récursive effectuant le branch and bound. A chaque étude d'un sous problème par ce branch and bound, l'ensemble bornant inférieur est enrichi des nouvelles solutions trouvées. Enfin, l'algorithme retourne l'ensemble bornant inférieur final.

3.2.1 La fonction récursive branch and bound

Voici une brève explication des paramètres de la fonction : le branch and bound opère le sondage des sous problèmes grâce à l'ensemble bornant inférieur, il faut donc que celui soit fourni. Un sous problème est caractérisé par un assignement de variables associé au problème. L'assignement du problème père est donc fourni. Enfin, la liste des points nadirs de l'ensemble bornant inférieur subit un traitement particulier expliqué dans une section future.

Algorithm 1: BranchAndBound

Data: Problème, Ensemble bornant inférieur, Variables assignées, Nadirs à prendre en compte
Result: Ajout de solutions à l'ensemble bornant inférieur

```
/* Le problème et l'assignement de variables représente un sous problème                    */
```

Calcul d'un ensemble bornant **inférieur** associé au sous problème;
Calcul d'un ensemble bornant **supérieur** associé au sous problème;

typeDominance ← **typeDeDominanceDuSousProbleme()**

if (typeDominance == optimal) ou (typeDominance == aucun) **then**
    mettre à jour ensemble bornant **inférieur**;
end

if (typeDominance == aucun) et (le sous problème n'est pas une feuille) **then**
    brancher en assignant la prochaine variable non assignée à 1;
    brancher en assignant la prochaine variable non assignée à 0;
end

Les variables assignées sont les variables du problème que les précédents branch and bound ont assignés à **0** ou à **1**.

3.2.2 Optimisation des points nadirs

Le test de dominance compare les points nadirs de l'ensemble bornant inférieur du problème global avec l'ensemble bornant supérieur du sous problème. Si l'ensemble bornant supérieur du sous problème passe au dessus d'un des points nadirs, alors le sous problème ne peut être sondé par dominance. On se place dans le cas d'un problème P1, si un point nadir se trouve au dessus de l'ensemble bornant supérieur de son sous problème P2, alors ce point nadir n'aide aucunement à sonder le sous problème par dominance. De plus, ce sera le cas pour tous les sous problèmes faisant partie du sous arbre de P2.

Pour optimiser le test de dominance, les points nadirs inutiles, ceux supérieurs à l'ensemble bornant supérieur du sous problème, sont donc supprimés de la liste des points nadirs à étudier.

Cette optimisation est implémentée dans la fonction branch and bound qui prend en paramètres les points nadirs utiles.

3.2.3 La mise à jour de l'ensemble bornant inférieur

Cette mise à jour récupère les nouvelles solutions non dominées trouvées dans le sous problème et les ajoute à l'ensemble bornant inférieur actuel. La mise à jour s'opère lorsque l'ensemble bornant supérieur du sous problème dépasse au moins un point nadir, c'est à dire à chaque fois qu'une nouvelle solution non dominée a pu être trouvée.

3.2.4 Heuristique primale

Dans le cas du problème du sac à dos dimensionnel, une heuristique primale a pour objectif de fournir à l'algorithme de résolution exacte un bon ensemble bornant inférieur de départ.

Dans l'implémentation, l'heuristique intervient à la suite d'une première dichotomie. Elle a pour objectif de trouver un maximum de points dans les triangles de l'ensemble bornant inférieur et ainsi améliorer les points nadirs.

Algorithme de recherche locale Les solutions considérées sont les points de l'ensemble bornant. A partir de chacune d'elle, une plus profonde descente est opérée utilisant pour voisinages des kp-échanges (1,1) et (0,1).

Fonction d'utilité Pour que la descente soit dirigée vers l'intérieur des triangles, la fonction d'utilité n'est pas l'usuelle comparaison euclidienne. La fonction utilisée favorise les solutions proches du milieu de l'hypoténuse de chaque triangle.

Chapitre 4

Manuel d'utilisation

Pour utiliser notre solver, il faut au préalable télécharger, notre code via le lien github : `https://github.com/LucasBaussay/BranchAndBoundBiObjKp`

La fonction main contient plusieurs paramètres booléens :

- withLinear : Si vrai, on utilise la relaxation linéaire lors de la dichotomie [default : false]
- withFirstConvex : Si vrai, on utilise l'enveloppe convexe caclulée avec JuMP dans la dichotomie [default : false]
- withHeuristic : Si vrai, on utilise l'heuristique primale avant l'appel au branch-and-bound [default : true]

De plus si rien n'est fixé, on fait appel à Combo pour la résolution lors de la dichotomie.

Une fois cela fait, en ouvrant un shell Julia dans le dossier *src*/ on peut :

```
>> julia
julia >> prob = Problem(nameInstance) #Pour charger un problème depuis un fichier
julia >> prob = Problem(nbVar) #Pour créer un problème aléatoire avec nbVar variables
julia >> main(prob) #Pour résoudre le problème avec Combo + Heuristique
julia >> main(prob, withLinear = true, withHeuristic = false) #Pour résoudre le problème avec
                                                    la relaxation linéaire sans l'heuristique
```

Chapitre 5

Améliorations et optimisations possibles

Comme on a pu le démontrer à l'aide des expérimentations, le meilleur algorithme est celui utilisant Combo avec l'heuristique primale présentée précédemment. Cependant ce code présente de nombreux problèmes, tout d'abord le temps d'exécution qui est encore trop élevé pour espérer résoudre des problèmes réels pouvant aller à plusieurs centaines voir des milliers de variables. Ainsi plusieurs pistes d'améliorations sont possible pour essayer de pallier à ce problème :

— Modifier le type d'algorithme branch-and-bound utilisé. L'actuel, le branch-and-bound global, arrive, de notre point de vue, aux limite de ce qu'il peut produire. En effet, en utilisant le meilleur algorithme possible pour la résolution exacte du sac-à-dos, les temps devraient idéalement être divisés par 10 au minimum pour être intéressant ;

— Une autre idée serait, tout en gardant un branch-and-bound global, d'utiliser l'amélioration de la relaxation linéaire, comme précisé précédemment, en effet celle-ci semble très prometteuse pour pouvoir rivaliser avec Combo ;

— Pour encore plus améliorer la relaxation linéaire, l'étude de différentes coupes est très prometteuse. En effet celle-ci, comme par exemples les coupes de Gomory, ont montrées une grande efficacité pour les branch-and-bound mono-objectif ;

— Enfin, cette étude se restreint sur encore deux spécificités. La première est le problème. On est ici limité au problème du sac-à-dos mono-dimensionnel, étudier un problème sac-à-dos plus général ou bien un problème non-structuré apporterait une difficulté supplémentaire. Malgré cela, ces problèmes sont nécessaire pour l'avancer des méthodes de résolution bi-objective qui sont encore beaucoup trop dominée par la méthode ϵ-contrainte ;

— Pour suivre le point précèdent, la limitation aux problème bi-objectif est elle aussi contraignante. L'étude des ensembles bornants pour des problèmes possédant trois voir plus d'objectifs reste un grand sujet de recherche. Comme on a put en parler dans ce papier, différentes méthodes ont été trouvés pour prolonger la méthode dichotomique à des problème multi-objectifs. Ainsi l'étude d'un branch-and-bound sur des problèmes multi-objectifs semble prometteuse, et pourrait même dans ce cas là dépasser l'ϵ-contraintes qui semble augmenter en complexité avec le nombre d'objectifs du problème ;

Chacun de ces points est une nouvelle étude possible, ce qui prouve à quel point les ensembles bornants sont un sujet majeur dans la recherche multi-objectifs. De plus on peut voir dans la partie définitions que les preuves, définitions et théorèmes sont encore très récents (le plus vieux date de 2001). Ce domaine de la recherche est en pleine expansion et les ensembles bornants joueront un rôle majeure dans les futurs découvertes.

Conclusion

Tout au long de ce travail, notre but aura été d'implémenter un algorithme de branch-and-bound, en approfondissant le concept d'ensembles bornants. En effet, c'est une notion fondatrice, qui permet l'étude de problèmes en multi-objectifs.

Nous avons sans cesse remis notre code en question pour essayer d'aller au bout de notre idée. Nous avons implémenter de nombreuses améliorations, plus ou moins efficaces. Mais chacune de ces pistes nous permettait de découler sur de nouvelles observations et de diversifier nos idées.

C'est ce que l'on peut voir dans les résultats présentés. Bien que les temps d'exécutions soient très élevés, ils ne font que diminuer au fur et à mesure de notre travail. Finalement, nous parvenons à résoudre des instances à 150 variables en moins de 10 minutes, là où nous peinions plus tôt à traiter un problème à 100 variables.

Malgré ces points positifs, nous n'arrivons pas à dépasser ce fameux seuil de 150 variables. On peut donc se questionner sur les limites de l'idée de base de notre algorithme, à savoir utiliser un branch-and-bound global. Une autre façon d'aborder le problème serait donc d'étudier chacun des triangles de la première enveloppe convexe. On pourrait ainsi effectuer un prétraitement sur chacun de ses triangles – le prétraitement global étant moins efficace. Cela permettrait potentiellement une convergence beaucoup plus rapide pour chaque triangle. Au cours de notre travail, nous avons constaté qu'un seul triangle bloque parfois le sondage d'un sous problème. Or, nous sommes obligés d'étudier tous les triangles à chaque fois, pour ne laisser aucun cas de côté. Travailler sur les triangles permettrait de sonder rapidement tous les triangles qui sont inutiles et de ne travailler que sur ceux qui apporteront de nouvelles solutions admissibles.

Pour encore améliorer la relaxation linéaire, un principe de coupe peut être mis en place pour calculer encore plus vite les solutions des sous-problèmes et essayer de converger vers des solutions entières plus rapidement.

Bibliographie

[1] Audrey Cerqueus. Bi-objective branch-and-cut algorithms applied to the binary knapsack problem. Theses, université de Nantes, November 2015.

[2] Librairie Combo. http://hjemmesider.diku.dk/~pisinger/combo.c.

[3] Matthias Ehrgott and Xavier Gandibleux. Bounds and Bound Sets for Biobjective Combinatorial Optimization Problems. 507 :241–253, 2001.

[4] Matthias Ehrgott and Xavier Gandibleux. Bound sets for biobjective combinatorial optimization problems. Computers Operations Research, 34 :2674–2694, 09 2007.

[5] Arthur M Geoffrion. Proper efficiency and the theory of vector maximization. Journal of Mathematical Analysis and Applications, 22(3) :618–630, 1968.

[6] Anthony Przybylski. Introduction à l'optimisation multi-objectif. 09 2020.

[7] Anthony Przybylski. Une idée simple pour une méthode de calcul pour la relaxation continue du problème de sac à dos mono-dimensionnel bi-objectif en variables binaires (rapport interne). 04 2021.

[8] Anthony Przybylski, Xavier Gandibleux, and Matthias Ehrgott. A two phase method for multi-objective integer programming and its application to the assignment problem with three objectives. Discret. Optim., 7(3) :149–165, 2010.

[9] Bernardo Villarreal and Mark H Karwan. An interactive dynamic programming approach to multicriteria discrete programming. Journal of Mathematical Analysis and Applications, 81(2) :524–544, 1981.

[10] Solveur vOptSovler. http://github.com/vOptSolver.