Etude de Resolution Search pour la programmation linéaire en variables binaires

Sylvain Boussier

UNIVERSITÉ MONTPELLIER II

— SCIENCES ET TECHNIQUES DU LANGUEDOC —

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ MONTPELLIER II

DISCIPLINE : **INFORMATIQUE**
Formation Doctorale : **Informatique - Mathématiques**
Ecole Doctorale : **Information Structures Systèmes**

SYLVAIN BOUSSIER

le 27 Novembre 2008

Titre :

Étude de Resolution Search pour la Programmation Linéaire en Variables Binaires

JURY

Gérard PLATEAU, Professeur, Université Paris XIII ..... Rapporteur
Dominique FEILLET, Professeur, École des Mines de Saint-Etienne ..... Rapporteur
Saïd HANAFI, Professeur, Université de Valenciennes ..... Examinateur
Nelson MACULAN, Professeur, Université de Rio de Janeiro ..... Examinateur
Olivier COGIS, Professeur, Université Montpellier II ..... Examinateur
Éric BOURREAU, Maître de Conférence, Université Montpellier II ..... Examinateur
Christophe WILBAUT, Maître de Conférence, Université de Valenciennes ..... Invité
Michel VASQUEZ, Enseignant chercheur - HDR, École des Mines d'Alès ... Directeur
Yannick VIMONT, Professeur, École des Mines d'Alès ..... Co-Encadrant

Remerciements

Le travail présenté dans ce mémoire a été effectué au Laboratoire de Génie Informatique et d'Ingénierie de Production de l'École des Mines d'Alès. Je tiens à exprimer toute ma gratitude et mon amitié à Messieurs Michel Vasquez, Directeur du centre de recherche L.G.I.2.P , et Yannick Vimont, Directeur de la recherche à l'École des Mines d'Alès, qui ont encadré mes travaux durant ces trois années de doctorat.

Mes remerciements vont ensuite à Messieurs Gérard Plateau, Professeur à l'université Paris XIII, et Dominique Feillet, Professeur à l'École des Mines de Saint-Etienne, pour l'intérêt qu'ils ont porté à cette étude et pour avoir accepté d'en être les rapporteurs.

Je tiens à remercier également l'ensemble des membres du jury, Messieurs Nelson Maculan, Professeur à l'université Fédérale de Rio de Janeiro, Saïd Hanafi, Professeur à l'université de Valenciennes, Olivier Cogis, Professeur à l'université Montpellier II, Éric Bourreau, Maître de Conférence à l'université Montpellier II et Christophe Wilbaut, Maître de Conférence à l'université de Valenciennes, pour avoir accepté de participer à la soutenance de ma thèse.

Je voudrais présenter des remerciements particuliers aux personnes, parfois non initiées aux termes techniques de l'optimisation combinatoire, qui ont pris le temps de relire et de corriger ce manuscript malgré la difficulté de l'exercice (dans l'ordre alphabétique) : Cyril Antonowicz, Ornella Debono, Pierre et Brigitte Fonda, Vandana Le Manchec, Françoise Scotti et Diana Marti Tatulli.

Je remercie enfin ma famille, mes parents, ma sœur Amandine (sans oublier Bob), les Mangrooviens : Adrien, Clémence, Ben, Clément, Nico, Peguy, Simon, Mathilde, le crew du Domaine : Romain, Blondin, Damien, Seb, Ben, Mika, Steph, Rémi, Marie, Cyril, Alex, Duc, Marie, Timothée (et oui...), l'ensemble du personnel du L.G.I.2.P et en particulier les thésards : Kamel, Sofiane, Wael, Gladis, Jorge, Sami, Abdelhak, Désiré et Saber. Je remercie encore une fois Ornella qui, en quelques sortes, a également passée sa thèse durant ces trois années.

À ma tante adorée...

Table des matières

Chapitre 1

Introduction

L'objectif de cette thèse est d'apporter une contribution à la résolution exacte de problèmes d'optimisation linéaire à variables binaires. Le fil conducteur en est l'étude de Resolution search [14] pour la résolution du problème du sac-à-dos multidimensionnel en 0–1. Cette introduction pose le contexte dans lequel se situent nos travaux et donne une vue générale des différents éléments qui seront présentés dans ce mémoire.

1.1 Contexte

La résolution de problèmes consistant à trouver l'optimum d'une fonction de variables entières soumises à un ensemble de contraintes a fait l'objet de nombreux travaux. Ces problèmes se formalisent de la manière suivante :

$$\begin{aligned} \text{Maximiser} \quad & f(x) \\ \text{sujet à} \quad & g_i(x) \leq b_i, \quad i = 1, ..., m, \\ & x \in \mathbb{Z}, \end{aligned} \tag{1.1}$$

où x est un vecteur de variables entières, f, g_i sont des fonctions réelles et b_i sont des valeurs réelles. Il existe de nombreuses classes de problèmes de ce type qui peuvent être obtenues en modifiant les propriétés des fonctions f et g_i. Le seul cas des problèmes d'optimisation linéaire à variables binaires, où f et g_i s'expriment de manière linéaire et où x est à valeurs 0 ou 1, permet de modéliser un grand nombre de problèmes industriels tels que la planification de tâches, le routage de véhicules, la gestion de stocks, etc.

De manière générale, la résolution de tels problèmes constitue une tâche difficile même pour des instances de taille moyenne[1]. En effet, le caractère discret des variables de décision et la présence des contraintes a pour conséquence que, contrairement au domaine continu des mathématiques, il n'existe pas d'équation générale de résolution pour une majorité de

[1] On considère qu'une instance est de petite taille en dessous de 30 contraintes / 100 variables, elle est de taille moyenne entre $5-10$ contraintes / 250 variables et 5 contraintes / 500 variables, sinon on parle d'instance de grande taille

ces problèmes. Ainsi, dans la plupart des cas, le seul algorithme de résolution consiste à énumérer chaque configuration du vecteur x, à vérifier qu'elle ne viole pas les contraintes, à calculer la valeur correspondante de la fonction objective et à mémoriser celle qui a la meilleure valeur. Un algorithme exhaustif de ce type est cependant limité à des instances de petite taille car il requiert un temps de calcul exponentiel en fonction du nombre de variables. Il existe de nombreuses techniques permettant d'éviter une énumération complète de l'espace de recherche et de réduire ainsi le temps de calcul nécessaire à la résolution d'instances de taille moyenne ; cependant, ces algorithmes d'énumération se heurtent en général à une combinatoire en $O(2^n)$, n étant la dimension du vecteur x.

On compte deux classes principales de méthodes de résolution pour les problèmes combinatoires de ce type : la première est constituée des méthodes dites exactes qui consistent à trouver une solution optimale au problème et la deuxième est constituée des méthodes dites approchées ou heuristiques qui n'explorent qu'une partie de l'espace de recherche dans le but de fournir la meilleure solution trouvée en un temps raisonnable. Dans ce dernier cas, rien ne garantit que les ensembles de configurations omis par cette recherche partielle ne contiennent pas une solution optimale, ce qui constitue le principal inconvénient de ces méthodes.

La voie que nous explorons ici consiste à concevoir des méthodes exactes capables de générer rapidement de bonnes solutions, c'est–à–dire de manière comparable à une heuristique reconnue performante sur le même problème. De telles méthodes assurent la convergence vers l'optimum tout en ayant les avantages des heuristiques car même si le processus est arrêté prématurément, il est possible d'obtenir une solution de qualité. Leur développement constitue cependant un challenge difficile car elles doivent être capables d'éliminer un maximum de configurations sous–optimales tout en favorisant l'obtention de bonnes solutions.

C'est dans cette optique que nous étudions Resolution search proposée par Chvátal en 1997. Cette méthode exacte, dédiée à la résolution de problèmes d'optimisation linéaire à variables binaires, présente un schéma d'exploration original qui intègre les concepts de backtrackings intelligents [2, 45] issus de la programmation par contraintes et de l'intelligence artificielle. L'ensemble de cette étude s'appuie sur la résolution du problème du sac à dos multidimensionnel en 0–1.

1.2 Problème du sac à dos multidimensionnel en 0–1

Dans cette section, nous définissons le problème et présentons une synthèse des méthodes de résolution existantes.

1.2.1 Définition du problème

Le problème du sac à dos multidimensionnel en 0–1, noté 01MKP pour *0–1 Multidimensional Knapsack Problem* (ou MKP en abrégé), permet de modéliser une grande variété de problèmes dans lesquels il s'agit de maximiser un profit tout en ne disposant que de ressources limitées. Au regard du nombre de travaux dont il a fait l'objet, à la quantité de problèmes concrets qu'il permet de modéliser et au nombre de jeux de données disponibles, nous pouvons considérer ce problème comme un problème de référence. Il peut être modélisé de la manière suivante :

$$\begin{aligned} \text{Maximiser} \quad & \sum_{j=1}^{n} c_j x_j \\ \text{sujet à} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq b_i, \quad i = 1, ..., m, \\ & x \in \{0,1\}^n \end{aligned}$$

avec $c_i \in \mathbb{N}$, $b_i \in \mathbb{N}$ et $a_{ij} \in \mathbb{N}$. Le MKP correspond à un programme linéaire en nombres entiers classique avec la particularité que la matrice des contraintes (matrice des coefficients a_{ij}) est composée de coefficients positifs et qu'il y a généralement peu de contraintes par rapport au nombre de variables. On rencontre le MKP dans de nombreux domaines d'application comme l'économie [48, 71, 49, 50], la gestion de stocks [34], l'industrie [17, 66], la gestion de chargements [5, 63] ou encore l'informatique répartie [32]. Les articles de Fréville [27] et de Fréville et Hanafi [24] présentent de nombreuses références à ce problème. Nous avons choisi de focaliser nos travaux sur ce problème car il représente toujours un challenge intéressant et il n'existe actuellement aucune approche permettant de résoudre des instances de taille moyenne à l'optimalité.

1.2.2 Jeux de données

La difficulté d'une instance de MKP est liée au nombre de variables et de contraintes qu'elle comporte mais également à la corrélation entre les profits c_j et les poids a_{ij} associés aux objets [56]. De nombreuses instances peuvent être trouvées sur le site de la `OR-Library`[2]. On y trouve des instances classiques (Petersen [55] ; Wiengartner et Ness [70] ; Senyu et Toyoda [62] ; Shih [63] ; Fréville et Plateau [25]) et des instances plus récentes (Chu et Beasley [4, 11]). Les instances classiques n'étant plus considérées comme difficiles aujourd'hui [11, 39], nous nous intéressons ici aux instances plus récentes proposées par Chu et Beasley qui sont devenues les instances de référence actuelles [67, 51, 6, 43, 68, 58, 73, 41, 72]. Ces instances ont été produites suivant la procédure proposée par Fréville et Plateau [29]. Les valeurs entières a_{ij} ont été générées aléatoirement entre 0 et 1000 et les valeurs des membres de droite on été générées en corrélation avec les a_{ij} et sont telles que $b_i = \alpha \sum_{j=1}^{n} a_{ij}$ où

[2] `http ://people.brunel.ac.uk/ mastjjb/jeb/orlib/mknapinfo.html`

$\alpha = 1/4, 1/2$ et $3/4$. Les coûts de la fonction objectif (c_j) sont générés en corrélation avec les a_{ij} et vérifient $c_j = \sum_{i=1}^{m} a_{ij}/m + 500d_j$ où d_j est un nombre aléatoire compris entre 0 et 1. Le site de la `OR-Library` contient des instances avec $n = 100$, 250 et 500 variables et $m = 5$, 10 et 30 contraintes. Trente problèmes ont été générés pour chaque (n, m) combinaison, donnant un total de 270 problèmes. Afin d'alléger l'écriture, on propose de nommer chacune de ces instances « cb**m**.**n**_**r** » ou **m** est le nombre de contraintes, **n** le nombre de variables et **r** le numéro de l'instance. Pour nommer, par exemple, l'ensemble des 30 instances à 10 contraintes et 500 variables, on parle de l'ensemble cb10.500 et la deuxième instance de cet ensemble sera notée cb10.500_2. Actuellement, les meilleures méthodes exactes ainsi que le solveur CPLEX[3] parviennent à prouver l'optimum pour les cb5.100, cb10.100, cb30.100, cb5.250 et cb5.500. Seulement 14 instances parmi les cb10.250 peuvent être résolues par CPLEX 9.2.

1.2.3 Méthodes de résolution

La complexité du MKP a encouragé le développement de méthodes approchées pour résoudre des instances ayant un grand nombre de variables. Ainsi, des métaheuristiques comme les algorithmes génétiques [11], la recherche locale tabou [39, 67, 68] ont été employées pour résoudre les instances à 500 variables du jeux Chu et Beasley. Actuellement, les meilleurs résultats connus sur ces instances ont été fournis par Vasquez et Vimont [68] et Wilbaut et Hanafi [72]. Deux états de l'art sur les méthodes de résolution existantes ont été proposés par Fréville [27] et Wilbaut *et al.* [74].

Les premières méthodes exactes utilisaient principalement la programmation dynamique [34, 70] mais ces méthodes ont une complexité spatiale trop importante qui les rend vite inapplicables pour des instances de grande taille [38]. Les algorithmes plus récents utilisent davantage le principe de l'énumération implicite (décrit en section 2.1.3 du chapitre 2). Cette méthode de résolution consiste à énumérer l'ensemble des configurations partielles du vecteur x en attribuant successivement différentes valeurs aux variables x_j. Chaque configuration visitée est évaluée en calculant une borne supérieure du sous–problème associé aux variables non–instanciées. Cette borne, qui peut être calculée de différentes manières, a pour but de réduire l'espace de recherche en éliminant une partie des configurations du vecteur x. Par exemple, Shih [63] propose de décomposer le problème en m problèmes de sac à dos simple en prenant en compte une seule des contraintes à la fois. Son algorithme est capable de résoudre des instances à 5 contraintes et 90 variables. Gavish et Pirkul [33] ont proposé des calculs de bornes basés sur les relaxations lagrangiennes, surrogates et composites. Leur algorithme présente de meilleurs résultats que celui de Shih [63], il est capable de résoudre des instances à 300 variables et 3 contraintes et des instances à 100 variables et 5 contraintes. Osorio *et al.* [52] ont développé une approche utilisant des

[3]CPLEX est un logiciel de résolution de problèmes d'optimisation édité par la société ILOG. Il est d'usage de comparer les performances des nouvelles méthodes de résolution avec celles de CPLEX pour évaluer leur qualité.

contraintes surrogates et des coupes logiques pour améliorer les performances de CPLEX. Leur algorithme est capable de résoudre les instances cb10.100 et certaines instances cb5.250 du jeu Chu et Beasley en 3 heures de temps de calcul. Plus récemment, James et Nakagawa [43] ont proposé une méthode utilisant la programmation dynamique pour générer des sous–problèmes dans lesquels certaines variables ont été fixées à 1. Certains de ces sous–problèmes sont éliminés par des règles de dominance et les sous–problèmes restants sont résolus par une énumération implicite. Leur algorithme est parvenu à obtenir les valeurs optimales des instances cb5.500 en un temps maximum de 7.79 heures. Ces valeurs optimales étaient alors inconnues.

Il existe également différentes techniques permettant de réduire l'espace de recherche. Il est possible, par exemple, de chercher un bon minorant[4] à l'aide d'une heuristique performante, de fixer certaines variables à leur valeur optimale [38, 26] ou encore de donner un encadrement du nombre d'objets à l'optimum [28].

1.3 Vue générale de la thèse

Dans le chapitre 2, nous présentons une énumération implicite pour résoudre le MKP. Cet algorithme intègre deux contraintes additionnelles afin de réduire la taille de l'espace de recherche. La première contrainte prend en compte la borne fournie par la relaxation continue, les coûts réduits des variables hors base et un minorant connu. Elle permet, d'une part, de fixer certaines variables hors base à leur valeur optimale et, d'autre part, de réduire l'espace de recherche. La deuxième contrainte fixe le nombre d'objets à une valeur k entière. La prise en compte de la contrainte $1 \cdot x = k$ dans le modèle réduit la borne supérieure donnée par la relaxation continue, ce qui permet de réduire davantage l'espace de recherche et d'augmenter le nombre de variables fixées par la contrainte des coûts réduits. De plus, elle permet de générer d'autres contraintes utiles pour résoudre de manière efficace des sous–problèmes ayant peu de variables. La politique de branchement de cette énumération implicite vise à explorer en priorité les affectations partielles susceptibles de violer la contrainte sur les coûts réduits. L'objectif est de favoriser la réduction de l'espace de recherche en explorant les configurations les moins « prometteuses » en priorité.

La méthode de résolution proposée utilise comme minorant de départ les solutions données par une heuristique de recherche locale performante (RL^{tabou} [67]). Elle permet ainsi de résoudre des instances de taille moyenne dans un temps raisonnable. En effet, les preuves d'optimalité manquantes des instances cb10.250 de Chu et Beasley sont obtenues et les instances cb5.500 sont résolues en un temps plus rapide que CPLEX et que l'algorithme de James et Nakagawa [43]. L'un des inconvénients majeurs de cette méthode est que sa politique de branchement visant à élaguer les nœuds de l'arbre de recherche au plus tôt la rend inefficace si aucun minorant de départ ne lui est fourni.

Dans le chapitre 3, nous présentons les différentes composantes de Resolution search

[4] On appelle *minorant* la valeur d'une solution réalisable dans le cas d'un problème de maximisation

et détaillons son déroulement sur un exemple simple. Des premières expérimentations menées sur des instances de MKP montrent l'intérêt de cette approche. Sa structure offre une souplesse dans l'exploration qui lui permet de trouver rapidement de bons minorants (elle trouve de meilleurs minorants plus rapidement que l'heuristique de recherche locale RL^{tabou}). En contrepartie, elle met plus de temps que l'énumération implicite du chapitre 2 pour résoudre l'ensemble des instances testées lorsqu'un bon minorant de départ est connu.

Dans le chapitre 4, nous proposons un algorithme exact combinant Resolution search avec une énumération implicite inspirée de celle du chapitre 2. Nous présentons certaines contributions à Resolution search, à savoir (i) une version itérative de l'algorithme permettant d'explorer pas à pas différents sous–problèmes de manière à favoriser la diversification de la recherche et (ii) l'intégration d'une nouvelle méthode d'identification des *obstacles*[5] minimaux fondée sur des relations d'implication qui peuvent être déduites entre les configurations partielles. Les expériences numériques réalisées démontrent les performances de cette approche hybride. Elle résout l'ensemble des instances cb5.250, cb10.250 et cb5.500 en un temps plus rapide que les meilleures méthodes exactes sur ces instances et elle permet de résoudre à l'optimum les instances cb10.500 dont les valeurs optimales étaient inconnues jusqu'à présent. Nous étudions également les possibilités d'amélioration de cette approche. Nous montrons par exemple qu'en resserrant la contrainte des coûts réduits, la réduction de l'espace de recherche qui en résulte permet de trouver de meilleurs minorants plus rapidement que les meilleures heuristiques connues sur les instances cb10.500.

Dans le chapitre 5, nous présentons une application de Resolution search à la résolution d'un problème industriel issu du sujet du challenge de la Société Française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF). Le sujet porte sur la planification de techniciens et d'interventions dans le domaine des télécommunications. Le problème est baptisé TIST pour *Technicians and Interventions Scheduling for Telecommunications*. Nous introduisons d'abord un algorithme de résolution fondé sur la métaheuristique GRASP pour la résolution approchée du TIST. Cet algorithme nous a permis d'être classé premier dans la catégorie *junior* et quatrième au classement général sur 17 équipes participantes. Dans une deuxième partie nous montrons qu'il est possible d'améliorer la performance globale de notre algorithme en résolvant un sous–problème du TIST par Resolution search.

Enfin, nous concluons ce mémoire et présentons des perspectives possibles à nos travaux dans le chapitre 6.

[5] Affectation partielle responsable d'un échec (voir chapitre 3).

Chapitre 2

Énumération implicite pour le sac à dos multidimensionnel en 0–1

Dans ce chapitre, nous présentons une énumération implicite pour résoudre le MKP. Cet algorithme est centré sur une analyse des coûts réduits à l'optimum de la relaxation continue du problème. Deux contraintes additionnelles sont intégrées au modèle original afin de réduire la taille de l'espace de recherche. La première contrainte prend en compte la borne fournie par la relaxation continue, les coûts réduits des variables hors base et un minorant connu. Elle permet, d'une part, de fixer certaines variables hors base à leur valeur optimale et, d'autre part, d'éliminer certaines configurations partielles du vecteur x. La deuxième contrainte fixe le nombre d'objets à une valeur k entière. Cette contrainte est issue d'une décomposition de l'espace de recherche en hyperplans ($1 \cdot x = k$, $k \in \mathbb{N}$) obtenue en utilisant un minorant calculé par une heuristique de recherche locale performante. La prise en compte de cette contrainte amène une diminution de la borne supérieure donnée par la relaxation continue du problème, ce qui permet de réduire davantage l'espace de recherche et d'augmenter le nombre de variables fixées par la contrainte des coûts réduits. De plus, elle permet de générer d'autres contraintes utiles pour résoudre de manière efficace des sous–problèmes ayant peu de variables. Contrairement à des méthodes plus classiques, la politique de branchement de cette énumération implicite vise à explorer en priorité les affectations partielles susceptibles de violer la contrainte sur les coûts réduits. Cette stratégie favorise l'élimination d'un maximum de configurations du vecteur x au plus tôt mais présente l'inconvénient de rendre la méthode dépendante de la qualité du minorant de départ. En termes de résultats numériques, cet algorithme résout les instances cb5.500 et cb10.250 du jeu Chu et Beasley de la `OR-Library`. La majorité des valeurs optimales des instances cb10.250 étaient inconnues jusqu'à présent. Le travail présenté dans ce chapitre à fait l'objet d'une publication [69].

2.1 Backtracking et énumération implicite

Dans cette section, nous introduisons le principe de l'énumération implicite, ainsi que quelques notations et définitions utiles pour le reste du mémoire.

2.1.1 Énumération exhaustive

Considérons le problème linéaire à variables binaires P suivant :

$$\begin{aligned} (P) \text{ Maximiser} \quad & f(x) \\ \text{sujet à} \quad & g_i(x) \leq b_i, \quad i = 1, ..., m, \\ & x \in \{0,1\}^n, \end{aligned}$$

avec f et g_i des fonctions linéaires positives ($f(x) \geq 0$ et $g_i(x) \geq 0$ pour $i = 1, ..., m$ quel que soit x) et b_i des valeurs entières positives ($b_i \geq 0$ pour $i = 1, .., m$). L'énumération exhaustive est la méthode la plus intuitive pour résoudre un tel problème combinatoire. Elle consiste à énumérer toutes les solutions, à les évaluer et à conserver celle donnant la meilleure valeur objective. On considère des *affectations partielles* $u \in \{0, 1, *\}^n$ des variables de décisions $x_1, ..., x_n$ telles que : $u_j = 1$ si x_j est instanciée à 1, $u_j = 0$ si x_j est instanciée à 0 et $u_j = *$ si x_j n'est pas instanciée (on dit alors que x_j est *libre*). Le problème P peut être résolu par l'algorithme d'énumération 2.1.

Algorithme 2.1 – Énumération exhaustive

```
enumeration_exhaustive(u,BI)
{
   if(u ∈ {0,1}^n) {
      for(i = 1, ..., m)
         if(g_i(u) > b_i) return;
      if(f(u) > BI) BI = f(u);
      return;
   }
   choisir un indice j tel que u_j = *;
   for(p = 1, 2) {
      u_j = φ_p;
      enumeration_exhaustive(u,BI);
   }
}
```

Cet algorithme prend en arguments un minorant `BI` (correspondant à la valeur d'une solution réalisable) et une affectation partielle u. Initialement, `BI`$= 0$ et $u = (*, *, ..., *)$. On définit ϕ le vecteur correspondant à l'ordre des valeurs choisies pour instancier les variables, on parle alors de *politique d'affectation*. Par exemple, $\phi = (1, 0)$ correspond à instancier d'abord les variables à la valeur 1 puis à la valeur 0. Lorsque toutes les variables sont

instanciées, la solution correspondante est évaluée. Si cette solution est réalisable et si sa valeur objective est supérieure à la meilleure valeur connue, le minorant `BI` est mis à jour. À la fin de la procédure, `BI` correspond à la valeur optimale de P.

Le processus de recherche revient alors à construire un arbre appelé *arbre de recherche* où chaque *nœud* de l'arbre est une affectation partielle. Tous les nœuds u^+ tels que $u_j^+ = u_j$ $\forall\ u_j \neq *$ sont appelés *nœuds descendants* de u, on dit également que u^+ est une *extension* de u.

2.1.2 Backtracking

L'énumération exhaustive est loin d'être la méthode la plus performante car elle explore les $2^{n+1} - 1$ nœuds de l'arbre de recherche. L'un de ses plus grands inconvénients est qu'elle ne vérifie la validité des contraintes qu'une fois toutes les variables instanciées. Le backtracking permet de mieux contrôler le processus d'exploration en identifiant plus tôt les affectations partielles qui ne satisfont pas les contraintes du problème ; on parle alors d'élagage de l'arbre de recherche. Considérons, par exemple le problème de sac à dos suivant :

$$\begin{aligned} (P') \text{ Maximiser} \quad & x_1 + x_2 \\ \text{sujet à} \quad & x_1 + x_2 \leq 1, \\ & x \in \{0,1\}^n, \end{aligned} \tag{2.1}$$

et sa résolution par l'algorithme 2.1 en choisissant à chaque nœud le plus petit indice j tel que $u_j = *$ et en considérant la politique d'affectation $\phi = (1, 0)$. La contrainte 2.1 rend l'affectation partielle $(1, 1, *, ..., *)$ irréalisable ; cependant le processus d'énumération exhaustive explore les 2^{n-2} combinaisons des variables $x_3, ..., x_n$ avant de remettre en cause l'affectation de x_2. Ce type de « pathologie » dans laquelle l'algorithme de recherche échoue de manière répétée pour les mêmes raisons est appelé *trashing*. Le *backtracking* [7, 10], défini par l'algorithme 2.2, a pour vocation de répondre en partie à ce problème. Il consiste à vérifier, à chaque nœud, la consistance de l'affectation partielle courante avec les contraintes du problème. L'unique différence entre le backtracking et l'énumération exhaustive est l'endroit où la vérification des contraintes est réalisée. Dans le cas où une violation de contrainte est identifiée, le backtracking change immédiatement la valeur de la dernière variable instanciée et continue le processus de recherche en considérant cette nouvelle valeur. Cette procédure permet de limiter le *trashing* et de réduire le nombre de nœuds visités. Par exemple, dans le cas de la résolution de P', lors de l'identification de l'inconsistance de $(1, 1, *, ..., *)$, la procédure fait un retour en arrière et visite directement la configuration $(1, 0, *, ..., *)$ au lieu d'énumérer les $2^{n-1} - 1$ nœuds descendants de $(1, 1, *, ..., *)$.

Algorithme 2.2 – Backtracking

```
backtracking(u,BI)
{
   for(i = 1, ..., m)
      if(g_i(u) > b_i) return ;
   if(u ∈ {0,1}^n) {
      if(f(u) > BI) BI = f(u) ;
      return ;
   }
   choisir un indice j tel que u_j = * ;
   for(p = 1, 2) {
      u_j = φ_p ;
      backtracking(u,BI) ;
   }
}
```

2.1.3 Énumération implicite

L'énumération implicite définie par l'algorithme 2.3 est une amélioration du backtracking. Elle intègre le calcul d'une borne supérieure à chaque nœud, ce qui lui permet de mieux élaguer l'arbre de recherche. Considérons une fonction d'évaluation `oracle` (selon la terminologie de [14]) définie par `oracle(`u`)`$= -\infty$ si $\bar{P}(u)$ est irréalisable et `oracle(`u`)`$= v(\bar{P}(u))$ sinon, avec $\bar{P}(u)$ la relaxation continue du problème $P(u)$ telle que :

$$
\begin{array}{lrll}
(P(u)) \text{ Max.} & f(x) & & \\
\text{s. à.} & g_i(x) \leq b_i, & i = 1, ..., m, & \\
& x_j = u_j & \text{si } u_j \neq * & \\
& x \in \{0,1\}^n & &
\end{array}
\qquad
\begin{array}{lrll}
(\bar{P}(u)) \text{ Max.} & f(x) & & \\
\text{s. à.} & g_i(x) \leq b_i, & i = 1, ..., m, & \\
& x_j = u_j & \text{si } u_j \neq * & \\
& x \in [0,1]^n & &
\end{array}
$$

et $v(\bar{P}(u))$ la valeur de la solution optimale de $\bar{P}(u)$. À chaque nœud, `oracle`(u) donne une borne supérieure de $P(u)$. Si `oracle`$(u) \leq$ `BI`, cela signifie qu'aucune extension de u ne mènera à une amélioration de la meilleure solution connue. Dans ce cas, la procédure effectue un retour en arrière.

Dans la pratique, l'énumération implicite (également appelée *branch & bound* [46]) est l'une des méthodes les plus couramment utilisées pour résoudre un problème d'optimisation à variables discrètes. Dans la plupart des cas, elle permet de réduire significativement le nombre de nœuds de l'arbre de recherche par rapport à un algorithme de backtracking. Nous présentons, dans ce chapitre, une méthode d'énumération implicite qui utilise certaines contraintes additionnelles et des règles de branchement spécifiques qui permettent d'améliorer l'algorithme standard pour la résolution du MKP.

Algorithme 2.3 – Énumération implicite

```
enumeration_implicite(u,BI)
{
   borne = oracle(u);
   if(borne ≤ BI) return;
   if(u ∈ {0,1}^n) {
      BI = borne;
      return;
   }
   choisir un indice j tel que u_j = *;
   for(p = 1,2) {
      u_j = φ_p;
      enumeration_implicite(u,BI);
   }
}
```

2.2 Contrainte des coûts réduits

L'utilisation des coûts réduits pour la fixation de variables et la réduction de l'espace de recherche n'est pas nouvelle en programmation linéaire. Nous pouvons citer, entre autres, les travaux de Saunders et Schinzinger [61], Balas et Martin [3], ceux de Fayard et Plateau [22] et plus récemment ceux d'Oliva *et al.* [51]. Nous montrons qu'en utilisant les coûts réduits à l'optimum de la relaxation linéaire du problème, nous pouvons générer une contrainte basée sur la borne supérieure donnée par cet optimum et sur un minorant quelconque. Cette contrainte permet de fixer certaines variables à leur valeur optimale et de réduire l'espace de recherche.

2.2.1 Description

Considérons P le MKP suivant :

$$\begin{aligned} (P)\ \text{Maximiser} \quad & c \cdot x \\ \text{sujet à} \quad & A \cdot x \leq b, \\ & x \in \{0,1\}^n, \end{aligned} \tag{2.2}$$

avec $c \in \mathbb{N}^n$, $b \in \mathbb{N}^m$ et $A \in \mathbb{N}^{m \times n}$, n étant le nombre de variables et m le nombre de contraintes. On considère $\bar{P}$ la relaxation linéaire de P. La résolution de $\bar{P}$ par la méthode du simplexe nécessite qu'il soit exprimé sous forme standard ; l'inégalité (2.2) doit être transformée en égalité en introduisant des *variables d'écart* à valeurs positives ou nulles, ce qui s'exprime de la façon suivante :

$$\begin{aligned}(\bar{P})\ \text{Maximiser} \quad & c \cdot x \\ \text{sujet à} \quad & A \cdot x + E \cdot s = b, \\ & x \in [0,1]^n, s \geq 0,\end{aligned}$$

où s est le vecteur des m variables d'écart et E la matrice des coefficients correspondante. Soit $(\bar{x}, \bar{s})$ la solution optimale de $\bar{P}$ et $\texttt{BS}$ sa valeur. Soit $(\bar{c}, \bar{u})$ le vecteur des coûts réduits correspondant aux variables (x, s) pour la solution de base $(\bar{x}, \bar{s})$. Le programme linéaire $\bar{P}$ peut également être écrit de manière équivalente comme suit :

$$\begin{aligned}(\bar{P})\ \text{Maximiser} \quad & \texttt{BS} + \sum_{j \in N^-} \bar{c}_j x_j - \sum_{j \in N^+} \bar{c}_j (1 - x_j) + \sum_{i \in M} \bar{u}_i s_i \\ \text{s. à.} \quad & \bar{A} \cdot x + \bar{E} \cdot s = \bar{b}, \\ & x \in [0,1]^n, s \geq 0,\end{aligned}$$

où $\bar{A}$, $\bar{S}$, $\bar{b}$ sont les valeurs correspondantes à la base associée à $(\bar{x}, \bar{s})$. M est l'ensemble des variables d'écart, N^- représente l'ensemble des variables hors base à leur borne inférieure et N^+ l'ensemble des variables hors base à leur borne supérieure. Si nous connaissons un minorant $\texttt{BI} \in \mathbb{N}$ de P, alors chaque solution de valeur meilleure que $\texttt{BI}$ doit satisfaire la contrainte suivante :

$$\lfloor \texttt{BS} \rfloor + \sum_{j \in N^-} \bar{c}_j x_j - \sum_{j \in N^+} \bar{c}_j (1 - x_j) + \sum_{i \in M} \bar{u}_i s_i \geq \texttt{BI} + 1$$

$$- \sum_{j \in N^-} \bar{c}_j x_j + \sum_{j \in N^+} \bar{c}_j (1 - x_j) - \sum_{i \in M} \bar{u}_i s_i \leq \lfloor \texttt{BS} \rfloor - \texttt{BI} - 1$$

Dans notre cas, nous ne tenons pas compte des variables d'écart. Á l'optimum, les variables d'écart sont positives et leurs coûts réduits sont négatifs, la somme $\sum_{j \in M} \bar{u}_i s_i$ est donc toujours négative. Nous pouvons donc enlever le terme $-\sum_{j \in M} \bar{u}_i s_i$ de l'inégalité ci–dessus. De plus, nous savons qu'à l'optimalité du programme linéaire, les variables hors base de coût réduit négatif sont égales à leur borne inférieure ($\bar{c}_j < 0 \Rightarrow x_j \in N^-$) et les variables hors base de coût réduit positif sont égales à leur borne supérieure ($\bar{c}_j > 0 \Rightarrow x_j \in N^+$). Nous pouvons donc considérer les valeurs absolues des coûts réduits au lieu des coûts réduits eux-mêmes. Par ce procédé, nous obtenons une contrainte que l'on appelle *contrainte des coûts réduits* :

$$\sum_{j \in N^-} |\bar{c}_j| x_j + \sum_{j \in N^+} |\bar{c}_j| (1 - x_j) \leq \lfloor \texttt{BS} \rfloor - \texttt{BI} - 1 \tag{2.3}$$

2.2.2 Fixation de variables

La contrainte des coûts réduits permet de fixer certaines variables hors base à leur valeur optimale. Par la suite, on définit la valeur *gap* de la manière suivante :

$$gap = \lfloor \texttt{BS} \rfloor - \texttt{BI} - 1$$

Si nous isolons une variable x_j hors base à l'optimum de $\bar{P}$ et considérons toutes les autres variables hors base à leur borne, c'est–à–dire à 0 pour les variables de coût réduit négatif et à 1 pour les variables de coût réduit positif, la contrainte (2.3) permet de déduire les relations suivantes :

$$\bar{x}_j = 1 \quad \Rightarrow \quad x_j \geq 1 - \frac{gap}{|\bar{c}_j|} \tag{2.4}$$

$$\bar{x}_j = 0 \quad \Rightarrow \quad x_j \leq \frac{gap}{|\bar{c}_j|} \tag{2.5}$$

où $\bar{x}_j$ est la valeur de x_j à l'optimum de $\bar{P}$ et $\bar{c}_j$ son coût réduit. En utilisant les relations (2.4) et (2.5), on obtient la proposition suivante :

Proposition 2.1 *Soit $\bar{P}$ la relaxation continue du MKP P. Soient $\bar{x}$ la solution optimale de $\bar{P}$, `BS` sa valeur et $\bar{c}$ les coûts réduits associés. Supposons que l'on connaisse un minorant `BI` de P, alors pour toute variable x_j hors base à l'optimum de $\bar{P}$,*

$$si\ |\bar{c}_j| > gap\ alors\ x_j\ doit\ être\ fixée\ à\ \bar{x}_j.$$

Connaissant un minorant $\texttt{BI} \in \mathbb{N}$ du problème, toute variable ayant un coût réduit de valeur absolue supérieure à *gap* est à $\bar{x}_j$ dans toute solution x de valeur meilleure que `BI`. Par exemple, considérons P^a le MKP suivant :

$$\begin{array}{llrcrcrcrcrcl}
(P^a) & \text{Maximiser} & 15x_1 & + & 10x_2 & + & 8x_3 & + & 8x_4 & + & 7x_5 & & \\
 & \text{sujet à} & 5x_1 & + & 2x_2 & + & 10x_3 & + & 1x_4 & + & 3x_5 & \leq & 10 \\
 & & 2x_1 & + & 11x_2 & + & 2x_3 & + & 10x_4 & + & x_5 & \leq & 11 \\
 & & x = & (x_1, x_2, x_3, x_4, x_5)^T \in \{0,1\}^5 & & & & & & & & &
\end{array}$$

La résolution de sa relaxation linéaire $\bar{P}^a$ par l'algorithme du simplexe nous donne une solution optimale $\bar{x}$ telle que

$$\bar{x} = (1, 0.717, 0.056, 0, 1)$$

un vecteur de coûts réduits $\bar{c}$ tel que

$$\bar{c} = (10.21, 0, 0, -0.56, 4.28)$$

et la valeur optimale $\texttt{BS} = 29.62$. Supposons que l'on connaisse un minorant $\texttt{BI} = 21$ du problème, la contrainte des coûts réduits correspondante s'écrit

$$10.21(1 - x_1) + 0.56x_4 + 4.28(1 - x_5) \leq 7$$

si l'on choisit $x_4 = 0$ et $x_5 = 1$ (ce qui minimise le membre de gauche) on a

$$0.31 \leq x_1$$

donc x_1 doit être fixé à sa valeur optimale $\bar{x}_1 = 1$ dans toute solution x telle que $c \cdot x > 21$.

2.2.3 Réduction de l'espace de recherche

La réduction de l'espace de recherche consiste à détecter des instanciations partielles des variables qui ne mènent pas à une amélioration de la meilleure solution connue (également appelées nœuds terminaux). Dans l'exemple précédent, l'affectation de x_1 à 0 ne mène à aucune solution de valeur supérieure à 21. C'est–à–dire que toutes les *extensions* de $(0, *, *, *, *)$ comme $(0, 1, *, *, *)$ ou $(0, *, 0, *, *)$ sont à exclure de l'espace de recherche. Par exemple, considérons P^b le MKP suivant :

$$\begin{array}{llrcrcrcrcrcr} (P^b) & \text{Maximiser} & 10x_1 & + & 10x_2 & + & 8x_3 & + & 8x_4 & + & 7x_5 & & \\ & \text{sujet à} & 5x_1 & + & 2x_2 & + & 10x_3 & + & 1x_4 & + & 3x_5 & \leq & 10 \\ & & 2x_1 & + & 11x_2 & + & 2x_3 & + & 10x_4 & + & x_5 & \leq & 11 \\ & & x = & (x_1, x_2, x_3, x_4, x_5)^T \in \{0,1\}^5 & & & & & & & & & \end{array}$$

La résolution de $\bar{P}^b$ nous donne un vecteur solution

$$\bar{x} = (1, 0.717, 0.056, 0, 1)$$

un vecteur de coûts réduits $\bar{c}$ tel que

$$\bar{c} = (5.21, 0, 0, -0.56, 4.28)$$

et la valeur optimale $\texttt{BS} = 24.62$.

Supposons que l'on connaisse un minorant $\texttt{BI} = 16$, la contrainte des coûts réduits s'écrit alors

$$5.21(1 - x_1) + 0.56x_4 + 4.28(1 - x_5) \leq 7$$

Contrairement à l'exemple précédent, on ne peut fixer aucune variable directement par la contrainte des coûts réduits car $\bar{c}_j \leq gap$ pour $j = 1, \ldots, 5$. Cependant, on peut identifier certaines instanciations partielles qui ne satisfont pas cette contrainte. Par exemple, l'instanciation $u = (0, *, *, *, 0)$ nous donne $0.56x_4 \leq -2.49$ or x_4 doit être positive donc u viole la contrainte des coûts réduits et par conséquent, tout vecteur v tel que $v_j = u_j$ si $u_j \neq *$ (v étant alors une *extension* de u) mène à une solution $c \cdot x \leq 16$ et doit être exclu de l'espace de recherche. On peut alors exprimer la contrainte des coûts réduits d'une manière différente. Soit $\Omega(u)$ l'ensemble des indices des variables x_j instanciées à l'opposé de leur valeur optimale (1 - $\bar{x}_j$) dans une instanciation partielle u, la somme des coûts réduits des variables x_j telles que $j \in \Omega(u)$ doit être inférieure ou égale à gap. Ceci nous donne la contrainte :

$$\sum_{j \in \Omega(u)} \bar{c}_j \leq gap. \tag{2.6}$$

Nous verrons que durant le processus d'énumération implicite, cette contrainte permet de déterminer l'insatisfiabilité (relativement à la contrainte $c \cdot x > \texttt{BI}$) de certaines affectations partielles.

2.3 Décomposition par hyperplans

L'encadrement du nombre d'objets contenus dans une solution optimale a été étudié par Glover [36] puis par Fréville et Plateau [28] et Vasquez et Hao [67]. Cela consiste à trouver deux valeurs entières k_{min} et k_{max} telles que $k_{min} \leq 1 \cdot x^* \leq k_{max}$, x^* étant une solution optimale du problème et 1 le vecteur unitaire. Supposons que l'on connaisse un minorant BI du problème, alors k_{min} est l'entier le plus proche supérieur ou égal à la valeur optimale du programme linéaire ($\bar{P}_{k_{min}}$) et k_{max} est le plus proche entier inférieur ou égal à la solution optimale du programme linéaire ($\bar{P}_{k_{max}}$) tels que :

$$\begin{array}{rrcrrc}
(\bar{P}_{k_{min}}) & \text{Minimiser} & 1 \cdot x & (\bar{P}_{k_{max}}) & \text{Maximiser} & 1 \cdot x \\
& \text{sujet à} & A \cdot x \leq b & & \text{sujet à} & A \cdot x \leq b \\
& & c \cdot x \geq \texttt{BI} + 1 & & & c \cdot x \geq \texttt{BI} + 1 \\
& & x \in [0,1]^n & & & x \in [0,1]^n.
\end{array}$$

En effet, soit z^*_{min} la valeur optimale du premier problème et z^*_{max} la valeur optimale du second problème. Si nous prenons moins de $\lceil z^*_{min} \rceil$ objets alors la contrainte $c \cdot x \geq \texttt{BI} + 1$ n'est pas satisfaite et si nous prenons plus de $\lfloor z^*_{max} \rfloor$ objets, l'une au moins des contraintes $A \cdot x \leq b$ n'est plus vérifiée.

Considérons à présent le problème $P(k)$ qui correspond au problème P avec la contrainte additionnelle $1 \cdot x = k \in \mathbb{N}$, définissant un hyperplan :

$$\begin{aligned} (P(k)) \text{ Maximiser} \quad & c \cdot x \\ \text{sujet à} \quad & A \cdot x \leq b, \\ & 1 \cdot x = k \qquad k \in \mathbb{N}, \\ & x \in \{0,1\}^n. \end{aligned} \tag{2.7}$$

Connaissant un minorant de P, il est possible de résoudre ce problème en résolvant indépendamment chaque problème $P(k)$. En effet, soit $\mathrm{E} = \{x \mid A \cdot x \leq b, x \in \{0,1\}^n\}$ l'espace des solutions admissibles de P et $\mathrm{E}_k = \{x \mid A \cdot x \leq b, c \cdot x > \mathtt{BI}, 1 \cdot x = k, x \in \{0,1\}^n\}$ l'espace des solutions admissibles de $P(k)$. Considérons $\mathrm{E}' = \{x \mid A \cdot x \leq b, c \cdot x > \mathtt{BI}, x \in \{0,1\}^n\}$ l'ensemble des solutions améliorant le minorant $\mathtt{BI}$, on a alors

$$\mathrm{E}' = \mathrm{E}_{k_{min}} \cup \cdots \cup \mathrm{E}_{k_{max}} \qquad \text{et} \qquad \mathrm{E}_{k_{min}} \cap \cdots \cap \mathrm{E}_{k_{max}} = \emptyset$$

L'ensemble $\{\mathrm{E}_{k_{min}}, \ldots, \mathrm{E}_{k_{max}}\}$ constitue une partition de E', on a donc

$$v(P) = max\{v(P(k)) \mid k_{min} \leq k \leq k_{max}\}.$$

Un intérêt de cette décomposition est que l'ajout de la contrainte $1 \cdot x = k \in \mathbb{N}$ au modèle original permet d'obtenir une borne supérieure $v(\bar{P}(k))$ généralement plus serrée que la borne supérieure donnée par $v(\bar{P})$. Cette diminution de la borne supérieure contribue à renforcer la fixation de variables et la réduction de l'espace de recherche induits par la contrainte des coûts réduits. Nous avons expérimenté l'influence cette décomposition sur une instance de MKP à 10 contraintes et 500 variables[1]. Pour cette instance, la résolution de la relaxation continue donne une borne supérieure de 304555.03 ce qui permet de fixer 30 variables par la contrainte des coûts réduits. Le tableau 2.1 expose les bornes supérieures et le nombre de variables fixées pour chaque hyperplan (colonne *# variables fixées*) dans le cas de la décomposition. Le minorant $\mathtt{BI} = 304214$ est donné par un algorithme glouton qui cherche à affecter $x_j = 1$ pour un nombre maximum d'objets j avec la plus grande valeur $\bar{x}_j$ dans une solution optimale donnée par la relaxation continue de P.

La première étape de l'algorithme consiste à partitioner l'espace de recherche en hyperplans en utilisant un minorant de départ (donné par l'heuristique de Vasquez et Hao [67]) puis à résoudre un à un les problèmes $P(k)$ correspondants. Durant l'exploration, le minorant est mis à jour si une meilleure solution réalisable est trouvée. L'ordre d'exploration des hyperplans est alors important car il influence l'amélioration du minorant. Partant de l'hypothèse qu'un hyperplan k ayant une grande valeur associée $v(\bar{P}(k))$ est plus « prometteur » qu'un hyperplan k' de valeur $v(\bar{P}(k'))$ plus faible, on propose d'explorer les hyperplans dans

[1] Cette expérience a été réalisée sur l'instance cb10.500_20 de la `OR-Library` (cf. chapitre 1).

TAB. 2.1 – Bornes supérieures et fixation de variables induite par $1.x = k$

hyperplan	borne sup.	# variables fixées
$1.x = 379$	304553.62	30
$1.x = 377$	304516.12	67
$1.x = 380$	304539.29	43
$1.x = 376$	304427.94	173
$1.x = 381$	304502.74	88
$1.x = 375$	304313.84	302
$1.x = 382$	304425.70	168
$1.x = 383$	304312.81	314

l'ordre décroissant des valeurs $v(\bar{P}(k))$ correspondantes. Bien que ce choix ne soit pas optimal, il s'avère expérimentalement intéressant comparé à d'autres heuristiques. Nous verrons en section 2.4 que la contrainte $1 \cdot x = k$ permet également de résoudre efficacement des sous–problèmes de petite taille par un algorithme de backtracking.

2.4 Procédure d'énumération implicite

Dans cette section, nous présentons l'évaluation des affectations partielles et la stratégie de branchement de notre énumération implicite. Nous détaillons ensuite un algorithme spécifique de propagation qui permet d'élargir la prise en compte des couts réduits aux parents du nœud courant. Nous exposons enfin l'algorithme de backtracking dédié à la résolution de sous–problèmes de petite taille.

2.4.1 Evaluation des affectations partielles

Une affectation partielle u est composée d'un sous–ensemble $F(u)$ des indices des variables fixées à 0 ou 1 et d'un sous–ensemble $L(u)$ des indices des variables libres. $F(u)$ est partitionné en deux sous–ensembles $F_0(u)$ et $F_1(u)$ contenant respectivement les indices des variables fixées à 0 et ceux des variables fixées à 1. Par exemple, une affectation partielle pourrait être la suivante :

$$u = (\overbrace{0,0,0}^{F_0(u)}, \overbrace{*,*,*}^{L(u)}, \overbrace{1,1,1}^{F_1(u)})$$

L'évaluation de u consiste à résoudre un sous–problème où les variables de décision sont les variables libres. En prenant en compte les variables fixées à 0 ou 1 dans l'affectation partielle et en considérant l'hyperplan k que nous sommes en train d'explorer, le sous–problème $P(k, u)$ est établi de la manière suivante :

$$(P(k,u)) \quad \text{Maximiser} \quad \sum_{j=1}^{n} c_j x_j \tag{2.8}$$

$$\text{sujet à} \quad \sum_{j=1}^{n} a_{ij} x_j \leq b_i, \qquad i = 1, ..., m, \tag{2.9}$$

$$\sum_{j=1}^{n} x_j = k, \tag{2.10}$$

$$x_j = u_j \qquad \text{si } u_j \neq *, \tag{2.11}$$

$$x_j \in \{0,1\} \qquad j = 1, ..., n. \tag{2.12}$$

On peut donc définir la fonction d'évaluation $\texttt{oracle}(k,u)$ de la manière suivante :

$$\texttt{oracle}(k,u) = \begin{cases} -\infty & \text{si } \bar{P}(k,u) \text{ est irréalisable} \\ v(\bar{P}(k,u)) & \text{sinon.} \end{cases}$$

Par la suite, on note $\texttt{BS}(u) = \texttt{oracle}(k,u)$ la borne supérieure à un nœud u donné.

2.4.2 Stratégie de branchement

À chaque nœud de l'arbre, la résolution du problème $\bar{P}(k,u)$ lors de l'appel de la fonction `oracle` nous donne la borne supérieure $\texttt{BS}(u)$ et l'information nécessaire à la génération de la contrainte des coûts réduits : le membre de gauche est donné par la solution optimale $\bar{x}^u$ sur les variables libres de u et les coûts réduits associés $\bar{c}^u$ tandis que le membre de droite est donné par la différence entre la borne supérieure $v(\bar{P}(k,u))$ et le minorant `BI` connu. Les variables x_j, $j \in L(u)$ sont ensuite triées par ordre décroissant de valeur absolue de coût réduit de manière à brancher en priorité sur des variables hors base à fort coût réduit. Par la suite, on définit $gap(u) = \lfloor \texttt{BS}(u) \rfloor - \texttt{BI} - 1$. Les variables x_j hors base de coût réduit $|\bar{c}^u_j| > gap(u)$ sont alors fixées à leur valeur optimale (cf. proposition 2.1) ce qui nous donne la règle de fixation suivante :

Règle de fixation 1 *À un nœud u donné, pour chaque variable x_j, $j \in L(u)$ hors base à l'optimum de $\bar{P}(k,u)$, si $|\bar{c}^u_j| > gap(u)$: fixer x_j à la valeur $\bar{x}^u_j$.*

Le branchement est ensuite réalisé sur la variable libre x_ϵ de plus fort coût réduit en la fixant à l'opposé de sa valeur optimale $(1 - \bar{x}^u_\epsilon)$ ce qui constitue le premier nœud fils u'. Une fois le problème $P(k,u')$ résolu, le branchement est fait en instanciant $u_\epsilon = \bar{x}^u_\epsilon$ *et* en fixant la prochaine variable libre hors base de plus fort coût réduit $x_{\epsilon'}$ à l'opposé de sa valeur optimale, ce qui constitue le deuxième nœud fils u''. Ce processus est réitéré jusqu'à ce que le nombre de variables libres soit inférieur ou égal à un paramètre *taille_spb*. Le sous–problème résiduel est résolu alors par un algorithme spécifique détaillé en section 2.4.4 que l'on note `dfs`. Puisque les branchements ne prennent pas en compte les variables de

base, *taille_spb* doit être supérieur ou égal au nombre de contraintes $(m+1)$. Ce principe d'énumération est décrit par l'algorithme 2.4 : la fonction `enumeration` prend en arguments une affectation partielle u, une valeur $k \in \mathbb{N}$ et un minorant `BI`.

Algorithme 2.4 – Algorithme d'énumération

```
enumeration(u,k,BI) {
   if(|L(u)| < taille_spb) {
      résoudre P(u) par dfs ;
      mettre BI à jour si une meilleure solution est trouvée par dfs ;
   }
   BS(u) = oracle(k, u) ;
   if(BS(u)≤ BI) return ;
   gap(u) = ⌊BS(u)⌋ − BI - 1 ;
   poser nhb = nombre de variables hors base ;
   trier les variables par ordre décroissant des |c̄_j^u| ;
   j_p = 1 ;
   while(|c̄_{j_p}^u| > gap(u)) j_p++ ;
   for(i = j_p, ..., nhb) {
      u_i = 1 − x̄_i^u ;
      for(j = 1, ..., i − 1) u_j = x̄_j^u ;
      enumeration(u,k,BI) ;
   }
   for(i = 1, ..., nhb) u_i = x̄_i^u ;
   enumeration(u,k,BI) ;
}
```

Par exemple, supposons que $u = (0, 1, *, *, *, *, *)$ et $\bar{x}^u = (\mathbf{0}, \mathbf{1}, 0, 1, 0, 0.6, 0.4)$ soit la solution optimale de $\bar{P}(k, u)$, x_4 et x_5 étant les variables de base. Supposons que les variables soient triées par ordre décroissant de valeur absolue des coûts réduits. Les nœuds générés sont alors les suivants :

$$(\mathbf{0}, \mathbf{1}, 1, *, *, *, *) \quad (\mathbf{0}, \mathbf{1}, 0, 0, *, *, *) \quad (\mathbf{0}, \mathbf{1}, 0, 1, 1, *, *) \quad (\mathbf{0}, \mathbf{1}, 0, 1, 0, *, *)$$

Proposition 2.2 *La résolution d'une instance de MKP par l'algorithme 2.4 est complète.*

Preuve Nous pouvons vérifier par un raisonnement par récurrence que ce type d'exploration est exhaustif. On pose $\alpha_j = \bar{x}_j$ et $\bar{\alpha}_j = 1 - \bar{x}_j$ et $p = n - taille\_spb$. La résolution d'une instance de MKP par l'algorithme 2.4 revient à résoudre un ensemble E^p de $p+1$ sous–problèmes tel que :

$$E^p = \{P(\bar{\alpha}_1, *, .., *), P(\alpha_1, \bar{\alpha}_2, *, .., *), ..., P(\alpha_1, .., \alpha_{p-1}, \bar{\alpha}_p, *, .., *), P(\alpha_1, .., \alpha_{p-1}, \alpha_p, *, .., *)\}$$

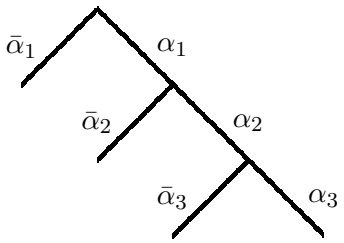
Nous allons montrer par récurrence que la résolution des $p+1$ sous–problèmes de l'ensemble E^p implique la résolution du problème initial $P(*, *, ..., *)$ quel que soit $p \leq n - (m+1)$.

La propriété est trivialement vraie au rang $p = 1$, montrons qu'elle est vraie au rang $p = 2$. Considérons l'ensemble E^2 suivant :

$$E^2 = \{P(\bar{\alpha}_1, *, ..., *), P(\alpha_1, \bar{\alpha}_2, *, ..., *), P(\alpha_1, \alpha_2, *, ..., *)\}$$

La résolution de $P(\alpha_1, \alpha_2, *, ..., *)$ et $P(\alpha_1, \bar{\alpha}_2, *, ..., *)$ par l'algorithme `dfs` résout $P(\alpha_1, *, *, ..., *)$ et la résolution de $P(\alpha_1, *, *, ..., *)$ et $P(\bar{\alpha}_1, *, *, ..., *)$ résout $P(*, *, *, ..., *)$, donc l'énumération est bien complète et la propriété est vraie au rang 2 (cf. figure 2.4.2).

Fig. 2.1 – Illustration de l'algorithme de branchement pour $p = 2$



Supposons que la propriété soit vraie au rang p, c'est–à–dire que la résolution des sous–problèmes de l'ensemble E^p implique la résolution du problème initial $P(*, *, *, ..., *)$. L'ensemble E^{p+1} est défini de la manière suivante :

$$E^{p+1} = \Big(E^p - \{P(\alpha_1, .., \alpha_p, *, .., *)\}\Big) \cup \{P(\alpha_1, .., \alpha_p, \bar{\alpha}_{p+1}, *, .., *), P(\alpha_1, .., \alpha_p, \alpha_{p+1}, *, .., *)\}$$

La résolution de $P(\alpha_1, ..., \alpha_p, \bar{\alpha}_{p+1}, *, ...*)$ et $P(\alpha_1, ..., \alpha_p, \alpha_{p+1}, *, ...*)$ par l'algorithme `dfs` résout $P(\alpha_1, ..., \alpha_p, *, ..., *)$, donc $E^{p+1} = E^p$. Par hypothèse, la résolution de tous les sous–problèmes de E^p implique la résolution du problème initial $P(*, *, .., *)$ donc la propriété est vraie au rang $p + 1$. □

Après avoir fixé la variable de plus fort coût réduit à sa valeur opposée, on constate qu'il est possible de fixer des variables supplémentaires. Comme expliqué en section 2.2, la contrainte 2.6, nous dit que la somme des coûts réduits des variables hors base fixées à l'opposé de leur valeur optimale doit être inférieure ou égale à la différence $gap(u)$. Cette valeur $gap(u)$ peut donc être vue comme une « quantité » de coût réduit disponible qui est consommée à chaque fois qu'une variable est fixée à sa valeur opposée $(1 - \bar{x}_j^u)$. A chaque branchement, une variable hors base est fixée à sa valeur opposée et la valeur $gap(u)$ est réduite du coût réduit $|\bar{c}_j|$ correspondant. Si $gap(u) < 0$, la contrainte 2.6 est violée donc $P(k, u)$ est irréalisable. Toute affectation partielle réalisable u doit donc satisfaire $u_j = \bar{x}_j^u$ pour x_j hors base et $|\bar{c}_j| > gap(u)$.

Règle de fixation 2 *Supposons que l'on fixe $x_j \in L(u)$, hors base à l'optimum de $\bar{P}(k,u)$, à la valeur $1-\bar{x}_j^u$. Pour chaque variable hors base x_i, tel que $i \in L(u)$ et $i \neq j$, si $|\bar{c}_j^u|+|\bar{c}_i^u| > gap(u)$, fixer x_i à la valeur $\bar{x}_i^u$.*

Cette règle de fixation supplémentaire (illustrée dans l'algorithme 2.8) permet de fixer davantage de variables hors base à chaque nœud.

Cette politique de branchement visant à fixer les variables hors base de fort coût réduit à l'*opposé* de leur valeur optimale est assez spécifique car elle oriente la recherche vers des zones de l'espace qui ne sont a priori pas favorables à l'obtention de « bonnes » solutions. En fixant les variables hors base à l'opposé de leur valeur optimale, nous aurons en effet davantage de chances de violer la contrainte des coûts réduits et donc de visiter une affectation partielle irréalisable. Dans la plupart des procédures par séparation et évaluation, le branchement se fait dans l'idée de chercher les zones « propices » de l'arbre de recherche. Or, dans notre cas, l'idée est au contraire, de chercher l'échec pour couper un maximum de nœuds au plus tôt. Nous verrons (cf section 2.6) que cette stratégie permet de résoudre des instances difficiles de MKP, mais que son efficacité est conditionnée par la connaissance d'un bon minorant de départ.

2.4.3 Propagation de la contrainte des coûts réduits

Lors de l'exploration d'un hyperplan $1 \cdot x = k$, toute affectation partielle u doit satisfaire la contrainte des coûts réduits donnée par la résolution du problème $\bar{P}(k,u)$ et le minorant `BI` connu. Nous proposons d'étendre cette vérification aux contraintes des coûts réduits générées par la résolution des problèmes $\bar{P}(k,u')$ associés aux nœuds parents u' de u. En effet, à une profondeur t de l'arbre de recherche, l'affectation partielle u doit satisfaire la contrainte des coûts réduits courante mais également toutes les contraintes générées aux nœuds parents de profondeur $t' < t$. On propose donc, à chaque nœud, de propager l'instanciation des variables dans u aux contraintes des nœuds parents et de vérifier la validité de celles-ci. Si une de ces contraintes est violée, l'affectation partielle courante est irréalisable et elle est exclue de l'espace de recherche.

Durant l'exploration, chaque fois qu'une contrainte sur les coûts réduits est générée, les valeurs optimales, les coûts réduits et la valeur *gap* correspondants sont mémorisés. Si à un nœud courant u, une variable x_j est instanciée à une valeur v et que cette même variable a une valeur optimale $\bar{x}_j^{u'} = 1 - v$ à un nœud parent u' de u, la valeur *gap* associée à u' est décrémentée de la valeur $\bar{c}_j^{u'}$ correspondante. Si $gap < 0$ pour une des contraintes des nœuds parents de u, alors u est considéré comme irréalisable et la procédure effectue un retour en arrière. Cette méthode permet de propager implicitement les coûts réduits sans ajouter explicitement de nouvelles contraintes au modèle. Nous considérons effectivement une seule contrainte qui est appliquée à chaque nœud (sous–problème) de l'arbre et la propagation est effectuée en mettant simplement à jour la valeur *gap* du nœud courant à la racine.

Les algorithmes 2.7 et 2.8 illustrent cette propagation des coûts réduits effectuée à chaque nœud de l'arbre de recherche. Les données `gap`, `cr` et `vo` correspondent à des tableaux de valeur tels que `gap[t]` $= gap(u)$, `cr[t]` $= \bar{c}^u$ et `vo[t]` $= \bar{x}^u$ au nœud u de profondeur t de l'arbre.

2.4.4 Algorithme de backtracking pour la résolution de sous–problèmes de petite taille

Lorsque le nombre de variables libres est inférieur à $taille\_spb$, le sous–problème résiduel est résolu par un algorithme de backtracking. Cet algorithme est limité à des problèmes de petite taille car il énumère les configurations de manière « brutale » sans résoudre $\bar{P}(k,u)$ à chaque nœud. En contrepartie, la contrainte $1 \cdot x = k$, $k \in \mathbb{N}$ permet de générer des contraintes additionnelles utiles pour couper certaines nœuds de l'arbre de recherche.

Lors de la résolution du problème $P(k,u)$, à un nœud u donné, La contrainte $1 \cdot x = k$ impose de choisir $k - |F_1(u)|$ variables à fixer à 1 parmi les variables libres. La somme des $k-|F_1(u)|$ variables libres de plus grand coût c_j est donc une borne supérieure du problème réduit aux variables libres de u. Par conséquent, si la somme des coûts des variables fixées à 1 dans u plus cette borne supérieure est inférieure au minorant `BI`, alors $P(k,u)$ est irréalisable selon la contrainte $c \cdot x >$ `BI`. On définit $\omega_c(j)$ l'ordre des variables $x_j \in L(u)$ par valeur décroissante de coût c_j, cette contrainte s'écrit :

$$\sum_{j \in F_1(u)} c_j + \sum_{j=1}^{k-|F_1(u)|} c_{\omega_c(j)} > \texttt{BI} \tag{2.13}$$

Par un même raisonnement, on peut déterminer une inégalité pour chaque contrainte de sac à dos. Si la somme des $k - |F_1(u)|$ variables libres de plus faible consommation a_{ij} plus la somme des consommations des variables fixées à 1 est supérieure à la capacité b_i, alors $P(k,u)$ est irréalisable selon la $i^{\text{ème}}$ contrainte de sac à dos. Si on définit $\omega_i(j)$ l'ordre des variables $x_j \in L(u)$ par valeur croissante de consommation a_{ij}, cette contrainte s'écrit :

$$\sum_{j \in F_1(u)} a_{ij} + \sum_{j=1}^{k-|F_1(u)|} a_{i\omega_i(j)} \leq b_i, \qquad i = 1, ..., m \tag{2.14}$$

On définit les fonctions $borne_c(u,k) = \sum_{j \in F_1(u)} c_j + \sum_{j=1}^{k-|F_1(u)|} c_{\omega_c(j)}$ et $borne_b(u,i,k) = \sum_{j \in F_1(u)} a_{ij} + \sum_{j=1}^{k-|F_1(u)|} a_{i\omega_i(j)}$ pour $i = 1, ..., m$ quels que soient u et k. Notons qu'il est possible de pré–calculer ces valeurs pour toute combinaison du nombre de variables libres et du nombre de variables à fixer à 1 dans le sous–problème. L'algorithme 2.5 décrit la méthode d'énumération proposée. Cet algorithme prend en paramètres : l'affectation partielle u, le tableau `idx` des indices absolus des variables libres de u, le vecteur b, l'hyperplan courant $1 \cdot x = k$, la profondeur t dans l'arbre de recherche (initialement $t = 0$), la somme des coûts des variables fixées à 1 dans u (notée *total* et initialisée à 0) et un minorant `BI`.

L'expérimentation nous a montré que cet algorithme n'est plus efficace pour des problèmes de taille $n > 20$. Par conséquent, la valeur du paramètre *taille_spb* doit respecter l'inégalité :

$$m + 1 \leq taille\_spb \leq 20.$$

Cette méthode de résolution est donc spécifiquement dédiée à la résolution de problèmes de MKP ayant moins de $m = 19$ contraintes. Il est possible de résoudre des problèmes ayant plus de contraintes en permettant le branchement sur des variables de base mais, dans ce cas, la méthode pert de son efficacité. Par exemple, les instances cb30.100 sont résolues en un temps comparable à celui de CPLEX alors que les instances cb10.100 et cb5.100 sont résolues plus rapidement. Dans la section 2.6.2, nous présentons une application de l'algorithme à la résolution des instances cb30.250 et cb30.500.

Algorithme 2.5 – Algorithme de backtracking pour des problèmes de petite taille

```
dfs(u,idx,b,k,t,total,BI) {
   if(borne_c(u,k) < BI) return;
   for(i = 1,...,m)
      if(borne_b(u,i,k) > b_i) return;
   if(|F_1(u)| == k) {
      if(total > BI) BI = total;
      return;
   }
   if(|F(u)| == n) return;
   if(k - |F_1(u)| < n - |F(u)|) {
      u_idx[t] = 0;
      dfs(u,idx,b,k,t+1,total,BI);
   }
   p =idx[t];
   u_p = 1;
   for(i = 1,...,m) {
      b'_i = b_i - a_ip;
      if(b'_i < 0) return;
   }
   dfs(u,idx,b',k,t+1,total + c_p,BI);
}
```

2.5 Algorithme général

Nous présentons l'algorithme global de résolution du MKP. L'algorithme 2.6 est le programme principal de résolution, il consiste à calculer les valeurs k_{min} et k_{max} à partir d'un minorant de départ et à exécuter l'énumération implicite sur chaque sous–problème $P(k)$ pour $k \in \{k_{min}, ..., k_{max}\}$ dans l'ordre décroissant des valeurs $v(\bar{P}(k))$ correspondantes.

L'énumération implicite est décrite par les algorithmes 2.8, 2.7 et 2.5. L'algorithme 2.7 constitue un programme auxiliaire qui consiste à créer le sous–problème à chaque nœud ainsi qu'à propager les coûts réduits aux nœuds parents. L'algorithme 2.8 décrit les différentes règles de branchement effectuées à chaque nœud en fonction des coûts réduits à l'optimum de la relaxation continue du sous–problème ainsi que l'appel récursif de l'algorithme 2.7. L'algorithme de backtracking est décrit par l'algorithme 2.5.

Algorithme 2.6 – Programme principal

```
resoudre_MKP(BI) {
   k_min = v(P̄_kmin) ; k_max = v(P̄_kmax) ;
   φ(k) = ordre de k suivant les valeurs v(P̄(k)) décroissantes ;
   for(k = k_min,...,k_max) {
      k' = φ(k) ;
      u = (*,*,...,*) ;
      enumeration_implicite(u,b,k',0,BI,gap,cr,vo) ;
   }
}
```

Algorithme 2.7 – Programme auxiliaire

```
programme_auxiliaire(u,b,k,t,BI,gap,cr,vo) {
   // Création du sous-problème
   b' = mise à jour des consommations b en fonction de u ;
   idx = indices absolus des variables libres de u ;
   total = somme des coûts c_j tels que u_j = 1 ;
   // Propagation de la contrainte des coûts réduits aux nœuds parents
   for(l = t,...,0) gap_[l] = gap[l] ;
   for(j = 1,...,n)
      for(l = t,...,0)
         if(u_j == 1 - vo[l]){
            gap_[l] = gap_[l] - cr[l] ;
            if(gap_[l] < 0) return ;
         }
   enumeration_implicite(u,b,k,t+1,BI,gap_,cr,vo)
}
```

Algorithme 2.8 – Algorithme d'énumération

```
enumeration_implicite(u,b,k,t,BI,gap,cr,vo){
   // Résolution du sous-problème par le backtracking
   if(|L(u)| ≤ 20) {
      définir idx le tableau des indices des variables libres de u ;
      dfs(u,idx,b,k,0,0,BI) ;
   }
   // Calcul de la borne supérieure et mémorisation de la contrainte des coûts réduits
   BS(u) = oracle(k,u) ;
   if(BS(u) ≤ BI) return ;
   else if(|F_1(u)| == k) {BI = BS(u) ;return ;}
   gap(u) = ⌊BS(u)⌋ − BI − 1 ;
   vo[t] = x̄^u ; cr[t] = c̄^u ; gap[t] = gap(u) ;
   //Création des problèmes fils
   nhb = nombre de variables hors base ;
   trier les variables par ordre décroissant des |c̄^u_j| ;
   j_p = 1 ;
   while(|c̄^u_{j_p}| > gap(u)) j_p++ ;
   for(i = j_p, ..., nhb) {
      u_i = 1 − x̄^u_i ;
      for(j = 1, ..., i − 1) u_j = x̄^u_j ;
      for(j = i + 1, ..., nhb)
          if(|c̄^u_i| + |c̄^u_j| > gap(u)) u_j = x̄^u_j ;
      programme_auxiliaire(u,b,k,t,BI,gap,cr,vo) ;
   }
   for(i = 1, ..., nhb) u_i = x̄^u_i ;
   programme_auxiliaire(u,b,k,t,BI,gap,cr,vo) ;
}
```

2.6 Résultats expérimentaux

Les tests expérimentaux ont été réalisés sur les instances du jeu Chu et Beasley de la `OR-Library` présentées dans le chapitre 1.

2.6.1 Preuves d'optimalité

Notre algorithme a été exécuté sur un P4 cadencé à 3.2 GHz avec 1 GB de RAM. Nos résultats sont comparés à ceux produit par le solveur CPLEX 9.2. Nous rappelons que le nom complet de chaque instance est cbm.n_r où m est le nombre de contraintes, n le nombre de variables et r le numéro de l'instance. Les tableaux 2.2, 2.3 et 3.6 détaillent respectivement les résultats obtenus sur les instances cb5.250, cb10.250 et cb5.500. La description des données par colonne est :

- *instance* : le nom de l'instance.
- `BI` : le minorant obtenu avec l'algorithme de recherche locale de Vasquez et Hao [67] pour cette instance.
- t^* : le temps de calcul en secondes nécessaire à l'algorithme de Vasquez et Hao pour trouver le minorant `BI`.
- z^{opt} : la solution optimale prouvée par notre algorithme.
- k^{opt} : le nombre d'objets dans la solution optimale.
- $diff$: l'écart entre le minorant et la solution optimale ($z^{opt} - \texttt{BI}$).
- t^{opt} : le temps requis en secondes par l'énumération implicite pour trouver la solution optimale (sans prendre en compte le calcul du minorant t^*).
- t^{total} : le temps total en secondes requis par notre algorithme ($t^* + t^{opt}$).
- t^{Cpx} : le temps requis en secondes par CPLEX 9.2 pour résoudre l'instance ou ERROR si un dépassement mémoire a eu lieu.

De nouvelles preuves d'optimalité ont été obtenues pour les instances à 10 contraintes et 250 variables (cf. tableau 2.3). Ces solutions étaient inconnues auparavant. CPLEX résout 47% des instances en moins de 4 heures, alors que l'énumération implicite proposée parvient à en résoudre 87% dans le même temps. La totalité des instances cb5.500 a été résolue en un temps inférieur à CPLEX et l'algorithme de James et Nakagawa [43] (qui ont été les premiers à publier les valeurs optimales de ces instances). Un des avantages de l'approche est qu'elle a permis d'effectuer de très longs calculs (par exemple 24 heures pour l'instance cb10.250_7) sans excéder la consommation de mémoire alors que CPLEX dépasse la capacité de mémoire après 4 heures. Notons que l'algorithme proposé est facilement parallélisable. Il est en effet possible d'explorer parallèlement chaque problème $P(k)$ avec $k \in \{k_{min}, \ldots, k_{max}\}$, ce qui permettrait de réduire le temps de calcul global. L'inconvénient principal de cette méthode de résolution est qu'elle est inopérante si aucun minorant de départ ne lui est fourni. A titre d'exemple, la résolution de l'instance cb5.100_0 requiert environ 100 secondes si aucun minorant de départ n'est fourni alors que CPLEX requiert seulement 6 secondes.

TAB. 2.2 – Résultats obtenus par l'énumération implicite sur les instances cb5.250

Instance	Minorant		Optimum					CPLEX
	BI	t*	z^{opt}	$diff$	k^{opt}	t^{opt}	t^{total}	t^{Cpx}
cb5.250_0	59312	80	59312	0	73	0	80	**33**
cb5.250_1	61472	79	61472	0	74	3	**82**	156
cb5.250_2	62130	61	62130	0	76	1	62	**11**
cb5.250_3	59453	106	59463	10	71	93	**199**	861
cb5.250_4	58951	95	58951	0	74	6	**101**	261
cb5.250_5	60062	78	60077	15	74	28	**106**	384
cb5.250_6	60396	98	60414	18	76	6	**104**	171
cb5.250_7	61449	101	61472	23	73	36	**137**	239
cb5.250_8	61885	73	61885	0	76	3	76	**69**
cb5.250_9	58959	87	58959	0	72	0	87	**16**
cb5.250_10	109086	96	109109	23	132	15	**111**	124
cb5.250_11	109841	97	109841	0	135	2	99	**40**
cb5.250_12	108508	72	108508	0	130	3	**75**	129
cb5.250_13	109378	98	109383	5	134	12	**110**	157
cb5.250_14	110718	95	110720	2	130	21	**116**	671
cb5.250_15	110256	91	110256	0	132	5	**96**	227
cb5.250_16	109040	79	109040	0	133	3	**82**	195
cb5.250_17	109042	95	109042	0	132	14	**109**	314
cb5.250_18	109971	100	109971	0	130	5	**105**	165
cb5.250_19	107058	90	107058	0	131	6	96	**18**
cb5.250_20	149659	91	149665	6	191	5	96	**101**
cb5.250_21	155940	92	155944	4	190	3	95	**62**
cb5.250_22	149334	95	149334	0	191	24	**119**	141
cb5.250_23	152130	103	152130	0	193	2	105	**94**
cb5.250_24	150353	98	150353	0	191	4	102	**90**
cb5.250_25	150045	73	150045	0	191	0	73	**9**
cb5.250_26	148607	75	148607	0	192	0	75	**6**
cb5.250_27	149772	95	149782	10	193	8	**103**	127
cb5.250_28	155061	91	155075	14	190	1	92	**17**
cb5.250_29	154662	95	154668	6	191	4	**99**	121

Tab. 2.3 – Résultats obtenus par l'énumération implicite sur les instances cb10.250

Instance	Minorant		Optimum					CPLEX
	BI	t*	z^{opt}	$diff$	k^{opt}	t^{opt}	t^{total}	t^{Cpx}
cb10.250_0	59187	205	59187	0	68	4921	**5126**	ERROR
cb10.250_1	58710	292	58781	71	69	43618	43910	**4369**
cb10.250_2	58094	174	58097	3	69	1335	**1509**	6746
cb10.250_3	60989	206	61000	11	70	8874	**9080**	ERROR
cb10.250_4	58068	276	58092	24	67	14487	**14763**	ERROR
cb10.250_5	58824	135	58824	0	68	964	**1099**	7394
cb10.250_6	58704	154	58704	0	67	898	**1052**	8255
cb10.250_7	58930	200	58936	6	69	87129	**87329**	ERROR
cb10.250_8	59382	188	59387	5	68	4240	**4428**	ERROR
cb10.250_9	59208	173	59208	0	69	6729	**6902**	ERROR
cb10.250_10	110913	180	110913	0	129	4772	**4952**	ERROR
cb10.250_11	108702	249	108717	15	127	7216	**7465**	ERROR
cb10.250_12	108932	190	108932	0	128	3192	**3382**	ERROR
cb10.250_13	110086	220	110086	0	131	14871	**15091**	ERROR
cb10.250_14	108485	184	108485	0	128	2122	**2306**	9869
cb10.250_15	110845	184	110845	0	130	5770	**5954**	ERROR
cb10.250_16	106077	263	106077	0	129	7604	**7867**	ERROR
cb10.250_17	106685	216	106686	1	128	5322	**5538**	ERROR
cb10.250_18	109822	251	109829	7	127	4566	**4817**	ERROR
cb10.250_19	106723	263	106723	0	131	1121	**1384**	5716
cb10.250_20	151801	203	151809	8	187	770	**973**	4695
cb10.250_21	148772	146	148772	0	188	2138	**2284**	ERROR
cb10.250_22	151900	175	151909	9	189	653	**828**	5048
cb10.250_23	151274	173	151324	50	189	5316	5489	**2234**
cb10.250_24	151966	179	151966	0	191	753	**932**	5727
cb10.250_25	152096	179	152109	13	189	639	**818**	2826
cb10.250_26	153131	154	153131	0	189	85	**239**	374
cb10.250_27	153563	210	153578	15	187	8259	**8469**	ERROR
cb10.250_28	149130	302	149160	30	187	974	**1276**	1638
cb10.250_29	149704	230	149704	0	190	596	**826**	2487

TAB. 2.4 – Résultats obtenus par l'énumération implicite sur les instances cb5.500

Instance	Minorant		Optimum					CPLEX
	BI	t*	z^{opt}	$diff$	k^{opt}	t^{opt}	t^{total}	t^{Cpx}
cb5.500_0	120114	480	120148	34	147	851	**1331**	8001
cb5.500_1	117844	597	117879	35	148	437	1034	**855**
cb5.500_2	121105	927	121131	26	144	185	**1112**	4687
cb5.500_3	120785	474	120804	19	149	156	**630**	6050
cb5.500_4	122291	406	122319	28	147	152	**558**	1578
cb5.500_5	121984	603	122024	40	153	329	**932**	3678
cb5.500_6	119117	680	119127	10	145	171	**851**	6937
cb5.500_7	120551	595	120568	17	150	209	**804**	2672
cb5.500_8	121566	513	121586	20	149	659	**1172**	ERROR
cb5.500_9	120663	776	120717	54	151	2007	**2783**	5756
cb5.500_10	218411	612	218428	17	267	226	**838**	1457
cb5.500_11	221118	584	221202	84	265	1562	2146	**905**
cb5.500_12	217523	842	217542	19	264	792	**1634**	3728
cb5.500_13	223534	837	223560	26	263	374	**1211**	5009
cb5.500_14	218966	474	218966	0	267	11	**485**	**485**
cb5.500_15	220520	679	220530	10	262	135	**814**	3122
cb5.500_16	219973	525	219989	16	266	68	**593**	1211
cb5.500_17	218194	670	218215	21	265	141	**811**	1096
cb5.500_18	216976	603	216976	0	262	94	**697**	2373
cb5.500_19	219689	612	219719	30	267	357	**969**	2522
cb5.500_20	295828	760	295828	0	383	7	767	**47**
cb5.500_21	308078	570	308086	8	384	104	674	**475**
cb5.500_22	299788	742	299796	8	385	41	783	**187**
cb5.500_23	306476	574	306480	4	384	46	**620**	1182
cb5.500_24	300340	543	300342	2	385	67	610	**204**
cb5.500_25	302565	607	302571	6	385	27	**634**	988
cb5.500_26	301327	439	301339	12	385	27	466	**366**
cb5.500_27	306430	496	306454	24	383	62	558	**148**
cb5.500_28	302809	640	302828	19	384	84	724	**367**
cb5.500_29	299904	425	299910	6	378	135	560	**478**

2.6.2 Nouveaux encadrements du nombre d'objets à l'optimum

Les instances cb30.250, cb10.500 et cb30.500 demeurent trop difficiles à prouver avec cette méthode. Cependant, même si le processus de recherche est stoppé prématurément, il peut fournir des résultats intéressants : (i) il peut fixer à leur valeur optimale certaines variables pour un hyperplan exploré et (ii) il peut prouver qu'aucune solution meilleure que le minorant donné n'est dans un hyperplan $1 \cdot x = k$. L'objectif ici est de réduire l'encadrement du nombre d'objets à l'optimum (donc de couper certains hyperplans). En considérant que la fonction $f(k) = v(\bar{P}(k))$ est convexe[2], nous proposons d'explorer les hyperplans en partant des valeurs extrêmes possibles de $[k_{min}, k_{max}]$ jusqu'aux valeurs centrales. L'objectif étant de prouver que pour un hyperplan $1 \cdot x = k$ il n'existe aucune solution $z >$ `BI`, nous explorons en premier lieu les hyperplans ayant les moins bonnes valeurs $v(\bar{P}(k))$.

Dans le tableau 2.5 sont donnés les nouveaux encadrements du nombre d'objets à l'optimum ($k_{min} \leq k^{opt} \leq k_{max}$) que nous avons obtenus pour les instances cb30.250, cb10.500 et cb30.500. Les valeurs en gras de k_{min}, k_{max} indiquent les nouvelles bornes trouvées par rapport aux bornes initiales obtenues avec le minorant fourni par l'algorithme de Vasquez et Vimont [68] (colonne *BI*). Nous avons réduit l'encadrement du nombre d'objets à l'optimum de 76% des instances dans un temps limite de 4 heures.

2.7 Conclusion

Nous avons montré que la prise en compte simultanée des coûts réduits à l'optimum de la relaxation continue et de la contrainte $1 \cdot x = k$ permet d'améliorer la fixation de variables et la réduction de l'espace de recherche. Nous avons proposé une méthode de branchement originale qui cherche des affectations partielles susceptibles de violer la contrainte des coûts réduits afin de couper au plus tôt un maximum de nœuds de l'arbre de recherche. Cette stratégie de branchement a pour conséquence de rendre l'approche efficace pour des instances moyennes (10 contraintes et 250 variables où 5 contraintes et 500 variables) dans le cas où un bon minorant est fourni (c'est–à–dire un minorant trouvé par une méthode heuristique performante par rapport à l'état de l'art connu). En revanche, cette même stratégie rend la méthode inopérante même pour des instances de petite taille (100 variables et 5 contraintes) dans le cas où aucun minorant de départ ne lui est fourni. Expérimentalement, notre algorithme a été testé sur les instances de référence de la `OR-Library`. Les résultats sont encourageants car les preuves d'optimalité des instances à 5 contraintes et 500 variables ont été obtenues en un temps plus rapide que les meilleurs résultats publiés sur ces instances et qu'avec le solveur commercial CPLEX. Les expérimentations sur les instances à 5 contraintes et 250 variables montrent une meilleure performance globale que CPLEX et de nouvelles preuves d'optimalité ont été fournies pour les instances à 10 contraintes et 250 variables de la `OR-Library`. L'algorithme présente l'avantage d'être facilement parallélisable.

[2]Cette propriété peut être prouvée en appliquant directement le théorème 10.2 du livre Linear Programming [12]

La distribution du calcul sur l'exploration des sous–problèmes $P(k)$ permettrait en effet d'accélérer le temps d'exécution. Cependant, un inconvénient est qu'il est particulièrement efficace pour des instances ayant moins de 19 contraintes. Une perspective serait d'améliorer son efficacité pour des instances à 30 contraintes, il faudrait pour cela, concevoir un algorithme performant pour la résolution de sous–problèmes à 30 contraintes et 30 variables ou modifier dynamiquement les règles de branchement dans le cas où le sous–problèmes n'est constitué que de variables de base.

TAB. 2.5 – Nouveaux encadrements du nombre d'objets à l'optimum

cb30.250				**cb10.500**				**cb30.500**			
r	BI	k_{min}	k_{max}	r	BI	k_{min}	k_{max}	r	BI	k_{min}	k_{max}
0	56824	**62**	**64**	0	117811	**134**	**136**	0	116056	128	133
1	58520	**65**	**66**	1	119232	**134**	**137**	1	114810	**126**	131
2	56553	**62**	**65**	2	119215	**135**	**137**	2	116712	**126**	**131**
3	56930	**63**	**65**	3	118813	**136**	**138**	3	115329	**126**	131
4	56629	**63**	**65**	4	116509	**133**	**138**	4	116525	**126**	**132**
5	57189	62	65	5	119504	**136**	**139**	5	115741	**129**	133
6	56328	**62**	**65**	6	119827	**138**	**139**	6	114181	**127**	**131**
7	56457	**63**	**65**	7	118329	**134**	**137**	7	114348	**126**	**131**
8	57442	62	66	8	117815	**136**	**137**	8	115419	126	132
9	56447	**63**	65	9	119231	**136**	**139**	9	117116	**126**	132
10	107770	**124**	**127**	10	217377	**256**	**259**	10	218104	**250**	254
11	108392	**124**	**126**	11	219077	**258**	**260**	11	214648	**250**	254
12	106399	**123**	**126**	12	217806	**255**	**258**	12	215978	**248**	**252**
13	106876	**124**	**126**	13	216868	**258**	**260**	13	217910	**250**	255
14	107414	**124**	**127**	14	213850	**255**	**260**	14	215689	**250**	254
15	107271	**125**	**127**	15	215086	**256**	**258**	15	215890	**251**	257
16	106365	**125**	**128**	16	217940	**259**	**261**	16	215907	**251**	255
17	104013	**124**	**127**	17	219984	**256**	**259**	17	216542	251	256
18	106835	**123**	126	18	214375	**256**	**258**	18	217340	251	256
19	105780	**125**	**127**	19	220899	**254**	**255**	19	214739	**251**	255
20	**150163**	**187**	**189**	20	304387*	**379**	**379**	20	301675	**374**	**377**
21	149958	**187**	**188**	21	302379	**380**	**380**	21	300055	**373**	379
22	153007	**187**	**188**	22	302416	**379**	**381**	22	305087	374	379
23	153234	**187**	**188**	23	300757	**379**	**380**	23	302032	**374**	**378**
24	150287	187	188	24	304374	**380**	**381**	24	304462	**375**	378
25	148574	**186**	**188**	25	301836*	**375**	**375**	25	297012	372	377
26	147477	**187**	**188**	26	304952	**377**	**379**	26	303364	372	377
27	152912	**186**	**188**	27	296478	**379**	**379**	27	307007	**375**	379
28	149570	**186**	**188**	28	301359	**379**	**380**	28	303199	374	379
29	149587	**186**	**188**	29	307089*	**378**	**378**	29	300572	**374**	378

Chapitre 3

Resolution search

Ce chapitre est consacré à l'étude de Resolution search [13, 14] ainsi qu'à son application à la résolution du MKP. Cette méthode de résolution, peu connue en recherche opérationnelle[1] a été conçue par Chvátal dans l'idée de proposer une alternative à l'énumération implicite, moins dépendante des stratégies de branchement. En effet, lors de la résolution d'un problème d'optimisation par l'énumération implicite, réaliser un branchement revient à *engager* la recherche dans la résolution d'un sous–problème. Lorsqu'une variable est fixée à une valeur, il est impossible de revenir en arrière tant que le sous–arbre correspondant n'a pas été entièrement exploré, c'est-à-dire que toutes les décisions de branchement prises dans ce sous–arbre n'ont pas été remises en cause une à une jusqu'à revenir au nœud courant. Ce caractère irrévocable rend les stratégies de branchement prépondérantes dans l'efficacité de l'énumération implicite. Afin de rendre les branchements plus pertinents et de réduire le *trashing* (voir chapitre 2, section 2.1), de nombreuses méthodes de l'intelligence artificielle ou de la programmation par contraintes analysent les causes des échecs rencontrés durant le processus de recherche, de manière à effectuer le retour en arrière directement sur les instanciations responsables des échecs. Ce type de backtracking, aussi appelé *backtracking intelligent*, reconsidère les choix de branchement dans un ordre plus approprié et pas strictement dans l'ordre chronologique comme c'est le cas dans l'énumération implicite. Resolution search utilise ces mécanismes de backtracking intelligent afin de considérer les branchements à titre indicatif et non à titre contraignant. Lorsqu'un échec est rencontré, l'affectation partielle ou complète en cause est mémorisée de manière à exclure le sous–arbre associé de l'espace des solutions et à déterminer rapidement *où* poursuivre la recherche. La mémorisation des échecs est gérée d'une manière spécifique qui permet de préserver l'espace mémoire tout en conservant la complétude. Dans ce chapitre, nous étudions de manière détaillée le fonctionnement de Resolution search ainsi que son application à la résolution du MKP.

[1] À notre connaissance, seuls les travaux de Hanafi et Glover [40], Demassey [20, 21], Palpant [53, 54] et Codato et Fischetti [15] mentionnent cette méthode.

3.1 Notations et définitions préalables

Extension : On définit une relation d'ordre partiel notée $\sqsubseteq$ entre deux affectations partielles v, w telle que

$$v \sqsubseteq w \quad \text{si} \quad w_j = v_j \quad \text{quelque soit} \quad v_j \neq *$$

Si $v \sqsubseteq w$, on dit que w est une *extension* de v.

Obstacle : On appelle *obstacle* une affectation partielle pour laquelle il est prouvé qu'aucune extension ne mène à une solution réalisable de valeur supérieure au minorant connu ou une affectation complète des variables. Par la suite, on représente les obstacles par des clauses.

Clause : On définit une clause comme un sous–ensemble parmi les symboles $x_1, x_2, ..., x_n$, $\bar{x}_1, \bar{x}_2, ..., \bar{x}_n$ également appelés *littéraux*. On dit que $u \in \{0, 1, *\}^n$ satisfait la clause C si $C \not\sqsubseteq u$. Un vecteur $(u_1, u_2, ..., u_n)$ est représenté par

$$\{x_j : u_j = 0\} \cup \{\bar{x}_j : u_j = 1\}.$$

Par exemple, l'affectation partielle $(*, 1, 0, *, 1, *, ..., *)$ est représentée par la clause $\bar{x}_2 x_3 \bar{x}_5$. La clause vide est notée $\emptyset$.

Résolvante : Deux clauses C_1 et C_2 sont dites en conflit s'il existe exactement un littéral w tel que $w \in C_1$ et $\bar{w} \in C_2$, leur *résolvante*, notée $C_1 \nabla C_2$, est définie par

$$C_1 \nabla C_2 = (C_1 - w) \cup (C_2 - \bar{w}).$$

La résolvante de deux clauses en conflit représentant des obstacles représente également un obstacle. Supposons par exemple que $u_1 = (1, 0, 1, *)$ et $u_2 = (1, 0, 0, *)$ soient des obstacles, ils peuvent alors être représentés par $C_1 = \bar{x}_1 x_2 \bar{x}_3$ et $C_2 = \bar{x}_1 x_2 x_3$ et simplifiés par l'obstacle $u_3 = (1, 0, *, *)$ correspondant à $C_1 \nabla C_2 = \bar{x}_1 x_2$.

3.2 Backtrackings intelligents et Resolution search

L'énumération implicite s'appuie sur le schéma d'exploration du backtracking qui remet systématiquement en cause les dernières variables instanciées sans regarder réellement les causes des échecs rencontrés. Prenons par exemple le problème P'' suivant :

$$
\begin{aligned}
(P'') \text{ Maximiser} \quad & x_1 + x_n \\
\text{sujet à} \quad & x_1 + x_n \leq 1, && (3.1)\\
& x_n \geq 1, && (3.2)\\
& x \in \{0,1\}^n.
\end{aligned}
$$

Supposons que l'on cherche à résoudre P'' par un algorithme de backtracking en instanciant les variables dans l'ordre lexicographique et en choisissant la politique d'affectation $\phi = (1, 0)$. L'algorithme fixe tout d'abord l'ensemble des variables à 1. Il détecte l'inconsistance de $(1, 1, ..., 1)$ et effectue un retour en arrière. Ensuite x_n est fixée à 0 pour obtenir le nouvel échec $(1, 1, ..., 0)$. Il remonte ensuite au niveau $n-1$, fixe x_{n-1} à 0 et essaie à nouveau d'instancier x_n à 1 puis à 0. L'algorithme déduit la contrainte $x_1 \neq 1$ uniquement lorsque tous les nœuds du sous–arbre correspondant à $x_1 = 1$ sont explorés. Il aurait été plus judicieux de se rendre compte immédiatement que les variables $x_2, .., x_{n-1}$ n'interviennent pas dans les contraintes du problème et de remettre directement en cause l'instanciation de x_1 à 1. Cette exploration inutile constitue un exemple de trashing.

Backjumping

S'il était possible de savoir quelle instanciation est responsable en cas d'échec, il serait intéressant de « remonter » directement au point où cette variable a été instanciée pour la fixer à une valeur différente. Ceci éviterait d'explorer inutilement toutes les combinaisons des autres variables. C'est dans cette idée que Gashnig [30, 31] a proposé le *backjumping*[2] (défini dans l'algorithme 3.1). La fonction `backjumping` prend en arguments : une affectation partielle u initialisée à $(*, *, ...*)$, une borne inférieure `BI` (initialisée à 0) et une clause S. Dans cet algorithme, le vecteur u est étendu tant qu'il ne viole pas les contraintes du problème. Si un échec est identifié, on mémorise dans l'obstacle S les instanciations des variables appartenant à une des contraintes violées. Si toutes les contraintes sont satisfaites mais que l'appel récursif de `backjumping` échoue, alors le nouvel obstacle S sera retourné par cet appel récursif. Si l'instanciation courante n'appartient pas à l'obstacle, alors la procédure fait directement un retour en arrière et va remonter ainsi l'arbre de recherche jusqu'à atteindre une variable de S. À la fin de la boucle, s'il n'y a aucune opportunité de saut en arrière, on génère l'obstacle Sc correspondant à l'ensemble des obstacles rencontrés dans les nœuds fils du nœud courant. Dans le cas où l'instanciation u est complète, on vérifie que u satisfait la contrainte $f(u) >$ `BI`, dans ce cas, S correspond à toutes les instanciations des variables de u ; dans le cas contraire S correspond aux instanciations des variables appartenant à la fonction objective f.

[2] L'idée du backjumping a été également introduite dans le dynamic branch & bound proposé par Glover et Tangedahl [37] (voir également Hanafi et Glover [40])

Algorithme 3.1 – Backjumping

```
backjumping(u,BI,S)
{
   if(u ∈ {0,1}){
      if(f(u) > BI) BI = f(u) ; S := instanciations dans u ;
      else S := instanciations des variables de u appartenant à f ;
      return ;
   }
   choisir un indice j tel que u_j = * ;
   Sc = ∅ ;
   for(p = 1,2) {
      u_j = φ_p ;
      if(toutes les contraintes sont respectées)
         backjumping(u,BI,S) ;
      else
         S := instanciations des variables de u violant une contrainte ;
      if(x_j ∉ S et x̄_j ∉ S )
         return ;
      else {
         if(x_j ∈ S) S = S − {x_j} else S = S − {x̄_j} ;
         Sc = Sc ∪ S ;
      }
   }
   S = Sc ;
   return ;
}
```

Supposons que l'on cherche à résoudre le problème P'' par le backjumping en choisissant à chaque nœud la variable libre de plus petit indice et en choisissant la politique d'affectation $\phi = (1, 0)$. Une première descente fixe toutes les variables à 1. Comme l'affectation $(1, 1, ..., 1)$ viole la contrainte 3.1, on identifie $S = \bar{x}_1\bar{x}_n$. Puisque $\bar{x}_n \in S$, on a $S = S - \{\bar{x}_n\}$. On continue et on explore $(1, 1, ..., 0)$ qui viole simultanément les contraintes 3.1 et 3.2 donc $S = \bar{x}_1 x_n$. L'instanciation $x_n \in S$ donc $S = S - \{x_n\}$ et $Sc = \bar{x}_1$. On en déduit que toute extension de $(1, *, ...*)$ n'améliore pas la borne inférieure. On effectue alors des retours en arrière successifs tant que l'instanciation courante n'appartient pas à S. Lorsque la procédure arrive au niveau 1, l'instanciation courante x_1 appartient à S donc on continue l'exploration en fixant x_1 à 0. Le backjumping est, sur cet exemple, nettement plus efficace que le backtracking car il remonte directement de l'instanciation $(1, 1, ..., 0)$ à l'instanciation $(0, *, ...*)$ sans passer par les nœuds intermédiaires des combinaisons des variables $x_2, ..., x_n$.

Learning

Le backjumping réduit le trashing par rapport au backtracking mais il ne l'élimine pas totalement. Prenons l'exemple du problème P''' suivant :

$$\begin{aligned} (P''') \text{ Maximiser} \quad & x_{n-1} + x_n \\ \text{sujet à} \quad & x_{n-1} + x_n \leq 1, \qquad (3.3) \\ & x_n \geq 1, \qquad (3.4) \\ & x \in \{0,1\}^n. \end{aligned}$$

Supposons que la politique de branchement de `backjumping` consiste à choisir la première variable libre dans l'ordre lexicographique et à poser $\phi = (1, 0)$. Lors de l'exploration de l'espace de recherche, `backjumping` essaie une première fois d'assigner la valeur 1 à x_{n-1} puis explore les deux configurations $(*, *, ..., 1, 1)$, $(*, *, ..., 1, 0)$. Dans les deux cas, il rencontre un échec et déduit que $S = \bar{x}_{n-1}$ est un obstacle, c'est à dire que toute solution contenant $x_{n-1} = 1$ est irréalisable. La procédure continue l'exploration sans avoir mémorisé cette nouvelle contrainte $x_{n-1} \neq 1$. Elle explore normalement les configurations $(*, *, ..., 0, 1)$ et $(*, *, ..., 0, 0)$ puis remonte au niveau $n - 2$. Une fois à ce niveau, x_{n-2} est fixée à 0 et la procédure reproduit de nouveau l'échec en fixant x_{n-1} à 1. Cette erreur est répétée 2^{n-2} fois et la procédure redécouvre à chaque fois que cela mène à des solutions irréalisables. L'objectif du *learning* est de mémoriser les obstacles rencontrés afin de prévenir les occurrences de ces mêmes obstacles dans les itérations futures. Ce principe a été introduit par Stallman et Sussman [64] dans la méthode *dependency-directed-backtracking* abrégée `DDB` (algorithme 3.2). La procédure `DDB` prend quatre arguments en paramètres : une affectation partielle u (initialement $u = (*, *, ...*)$), une borne inférieure `BI` (initialisée à 0), une clause S et un ensemble $\mathcal{F}$ d'obstacles.

`DDB` est similaire au backjumping avec la différence que les obstacles sont mémorisés au cours de la recherche. À chaque nœud, lorsque tous les nœuds descendants ont été explorés, l'obstacle courant est mémorisé dans $\mathcal{F}$. À chaque itération, on vérifie alors que l'affectation partielle courante satisfait non seulement les contraintes du problème mais également la liste des obstacles de $\mathcal{F}$ (rappelons qu'une affectation partielle u satisfait un obstacle S si $S \not\sqsubseteq u$). Si l'affectation courante ne satisfait pas un des obstacles mémorisés, ce sont les instanciations de cet obstacle qui sont prises en compte pour construire S. Dans l'exemple précédent, la résolution de P''' par `DDB` explore 2^{n-1} configurations en moins qu'avec `backjumping`. En effet, une fois la contrainte $x_{n-1} \neq 1$ déduite, elle est mémorisée dans $\mathcal{F}$ sous forme d'obstacle $S = \bar{x}_{n-1}$. De cette façon, chaque fois que la variable x_{n-1} est instanciée à 1 dans le nœud u courant, la procédure effectue un retour en arrière car dans ce cas u est une extension de S. La procédure `DDB` élimine en grande partie le trashing mais au prix d'une consommation mémoire importante ; à chaque nœud, `DDB` mémorise un obstacle, sa complexité spatiale peut donc être exponentielle suivant le problème traité.

Algorithme 3.2 – Dependency-Directed Backtracking

```
DDB(u,BI,S,F)
{
   if(u ∈ {0,1}){
      if(f(u) > BI) BI = f(u); S := instanciations dans u;
      else S := instanciations des variables de u appartenant à f;
      return;
   }
   choisir un indice j tel que u_j = *;
   for(p = 1,2) {
      u_j = φ(p);
      if(toutes les contraintes sont respectées et u ⋢ C pour tout C ∈ F)
         DDB(u,BI,S,F);
      else
         S := instanciations des variables de u violant une contrainte ou un obstacle;
      if(x_j ∉ S et x̄_j ∉ S )
         return;
      else {
         if(x_j ∈ S)  S = S − {x_j} else  S = S − {x̄_j};
         Sc = Sc ∪ S;
      }
   }
   S = Sc;
   F = F ∪ {S};
}
```

K-order learning

Afin de palier à la demande de consommation mémoire du learning, Dechter a proposé le *k-order learning* [19]. Cette méthode, similaire à DDB, ne mémorise que les obstacles de taille inférieure à une valeur k donnée. La complexité spatiale de DDB devient alors polynomiale. En effet, puisque nous avons n variables et un domaine maximum de taille 2 pour chaque variable, le nombre total d'obstacles de taille inférieure à k mémorisés est :

$$C_n^0 + 2 \cdot C_n^1 + 2^2 \cdot C_n^2 + ... + 2^{k-1} \cdot C_n^{k-1} + 2^k \cdot C_n^k.$$

Dynamic backtracking

Le *dynamic backtracking* de Ginsberg [35] (noté DBT) est également une méthode qui exploite le backjumping et le learning tout en ayant une complexité spatiale polynomiale. Le principe de DBT est de supprimer les obstacles qui ne sont plus valides avec l'instanciation courante afin de réduire la consommation de mémoire. Comme les algorithmes précédents, DBT commence par une instanciation vide des variables $u = (*, *, ...*)$ et étend progressive-

ment cette instanciation jusqu'à explorer (implicitement) la totalité de l'espace de recherche. Comme dans le cas de DDB, durant la procédure, DBT génère des obstacles qui excluent des portions de l'espace de recherche. Lorsque l'obstacle $S = \emptyset$ est généré, cela signifie qu'aucune solution réalisable n'améliore la borne inférieure BI et que, par conséquent, BI est la valeur optimale du problème. Cet algorithme permet de limiter la consommation de mémoire en « oubliant » certains obstacles jugés non-pertinents. Pour ce faire, DBT considère des *explications* du retrait des valeurs de certaines variables. Une explication est constituée d'un obstacle S et d'un littéral w appartenant à S. Ce littéral correspond à la dernière instanciation qui a causé l'échec. Par exemple, si l'obstacle $S = x_1\bar{x}_2x_3$ (correspondant à $(0, 1, 0, *, ..., *)$) est identifié suite à l'instanciation de x_3 à 0, on détermine un littéral $w_S = x_3$ signifiant que $x_1\bar{x}_2 \Rightarrow x_3$ (autrement dit : $x_1 = 0$ et $x_2 = 1$ impliquent $x_3 \neq 0$).

On considère un ensemble $\mathcal{F}$ de clauses $C_1, C_2, ..., C_N$ et un ensemble de littéraux associés $w_1, w_2, ...w_N$ constituant un ensemble d'explications. Durant le processus de recherche, seuls les explications jugés valides sont conservés dans $\mathcal{F}$. On dit qu'une explication $\{C_i, w_i\}$ est valide par rapport à u si $C_i - \{w_i\} \sqsubseteq u$. Par exemple, $\{C = x_1\bar{x}_2\bar{x}_4, w = \bar{x}_4\}$ est valide par rapport à $u = (0, 1, 0, *, ...*)$. L'ensemble des w_i pour $i = 1, ...N$ est un ensemble d'instanciations interdites. Lors de l'instanciation de nouvelles variables, on vérifie qu'elles n'appartiennent pas à l'ensemble $\{w_1, ..., w_N\}$. Si deux obstacles contiennent les instanciations x_i et $\bar{x}_i$ d'une variable, on réduit ces obstacles par résolvante. Ensuite, on supprime de $\mathcal{F}$ tous les obstacles qui mentionnent la variable x_i (car chaque instanciation de x_i viole un obstacle). Supposons que DBT instancie les variables dans l'ordre lexicographique et que les explications $\{C_1 = x_1\bar{x}_2x_5, w_1 = x_5\}$ et $\{C_2 = x_1\bar{x}_3\bar{x}_5, w_2 = \bar{x}_5\}$ soient générées. On peut alors remplacer C_1 et C_2 par leur résolvante $R = C_1 \nabla C_2 = x_1\bar{x}_2\bar{x}_3$. Le littéral w associé à R doit ensuite correspondre à la *dernière* instanciation réalisée sinon DBT peut cycler indéfiniment [35]. Dans notre cas, on choisit $w_R = x_3$. Cette dernière étape revient à effectuer un backjumping de la variable x_5 à x_3. La procédure DBT a une complexité spatiale polynomiale car un seul obstacle (ou explication) est mémorisé pour chaque couple (variable,valeur). Il y a donc au plus $2n$ obstacles en mémoire, chacun ayant dans le pire des cas $(n-1)$ littéraux, ce qui nous donne une complexité spatiale en $O(2n^2)$.

Resolution search

Resolution search proposée par Chvátal [14] est une méthode de résolution proche du dynamic backtracking. Comme DBT, Resolution search exploite à la fois le principe du backjumping et du learning tout en ayant une complexité spatiale polynomiale. Nous introduisons ici son fonctionnement général, une étude plus détaillée est présentée dans les sections qui suivent.

Resolution search est un processus itératif qui alterne successivement la recherche d'une solution et la mémorisation d'obstacles correspondants aux échecs rencontrés. La recherche de solutions est réalisée par une fonction `obstacle` qui effectue deux phases distinctes : une

phase de descente (ou *waxing phase*) qui attribue des valeurs 0 ou 1 à l'affectation partielle courante jusqu'à obtenir soit un échec, soit une solution réalisable, puis une *phase de remontée* (ou *waning phase*) qui réduit pas à pas l'affectation partielle courante de manière à obtenir un obstacle minimal. Comme dans le cas de `DBT`, les obstacles sont mémorisés dans un ensemble (appelé famille $\mathscr{F}$) comprenant un ensemble de N clauses C_1, C_2, ..., C_N et un ensemble de littéraux associés w_1,w_2,...,w_N. Le choix de w_i est soumis à certaines contraintes mais, contrairement à `DBT`, il ne correspond pas nécessairement à la dernière instanciation réalisée et peut être choisi plus librement. La famille $\mathscr{F}$ possède une structure particulière appelée *path-like* qui impose néanmoins certaines règles lors de l'insertion des obstacles et lors du choix des w_i. Cette structure path-like permet à Resolution search d'avoir une complexité spatiale en $O(n)$ tout en ayant une convergence finie. De manière analogue à `DBT`, Resolution search considère les littéraux w_i comme des instanciations « interdites » pour les variables libres. Cependant, au lieu de considérer simplement le caractère prohibitif de ces instanciations, Resolution search les utilise pour « guider » la recherche. L'espace de recherche est exploré d'une manière plus souple car, lorsqu'un échec est identifié, le retour ne se fait pas strictement sur la dernière instanciation mais il est effectué simultanément sur l'ensemble des obstacles mémorisés : après chaque itération, le prochain nœud de l'exploration est obtenu en faisant l'union des $(C_i - \{w_i\}) \cup \{\bar{w}_i\}$ pour $i = 1, ..., N$. Par exemple, si à une itération donnée, nous avons

$$\begin{aligned} \mathscr{F} := \quad & C_1 = x_1 && (w_1 = x_1), \\ & C_2 = x_2 x_3 && (w_2 = x_2), \\ & C_3 = \bar{x}_2 x_3 \bar{x}_4 \bar{x}_5 && (w_3 = \bar{x}_4), \end{aligned}$$

le nœud suivant dans l'exploration est le nœud $u = (1, 1, 0, 0, 1, *, ...*)$ (donné par la clause $\bar{x}_1 \bar{x}_2 x_3 x_4 \bar{x}_5$). Ce nœud n'est l'extension d'aucun des obstacles mémorisés[3]. Lors de l'ajout d'un nouvel obstacle S à $\mathscr{F}$, Resolution search tente également de simplifier S en effectuant des résolvantes de S avec les différents obstacles de $\mathscr{F}$. La fonction `resolution_search` est décrite par l'algorithme 3.3, elle prend en paramètre un minorant noté `BI`. Les différentes fonctions de cet algorithme sont détaillées dans les sections qui suivent.

[3]Il est intéressant de noter que `DBT` et Resolution search ont un fonctionnement symétrique : `DBT` ne conserve que les obstacles qui satisfont l'affectation partielle courante alors qu'en opposition, Resolution search cherche des affectations partielles qui satisfont les obstacles courants.

Algorithme 3.3 – Resolution search

```
resolution_search(BI)
{
   u = (*, *, ..., *) ;
   while(1) {
      try = obstacle(u,BI,S) ;
      if(try > BI)  BI = try ;
      mise_a_jour(F,S) ;
      u = u(F) ;
      if((*,*,...,*) ∈ F)   break ;
   }
}
```

3.3 Fonction oracle

La fonction d'évaluation des affectations partielles `oracle` est définie par la résolution du problème $\bar{P}(u)$ suivant :

$$(\bar{P}(u)) \quad \text{Maximiser} \quad \sum_{j=1}^{n} c_j x_j \tag{3.5}$$

$$\text{sujet à} \quad \sum_{j=1}^{n} a_{ij} x_j \leq b_i \qquad i = 1, ..., m \tag{3.6}$$

$$x_j = u_j \qquad \text{si } u_j \neq * \tag{3.7}$$

$$x_j \in [0, 1] \qquad j = 1, ..., n \tag{3.8}$$

Soit $v(\overline{P}(u))$ la valeur optimale de $\bar{P}(u)$, on a

$$\texttt{oracle}(u) = \begin{cases} -\infty & \text{si } \overline{P}(u) \text{ est irréalisable} \\ v(\overline{P}(u)) & \text{sinon} \end{cases}$$

Durant le processus d'exploration, on note `BI` la valeur de la meilleure solution connue (minorant). Si $\texttt{oracle}(u) \leq \texttt{BI}$ alors u est un obstacle.

3.4 Fonction obstacle

Partant de l'affectation partielle courante u, la fonction `obstacle` explore partiellement l'espace de recherche en exécutant deux phases distinctes : (1) La *phase de descente* (ou *waxing phase*) qui consiste à étendre le nœud u en u^+ en remplaçant successivement les éléments $u_j = *$ par 0 ou 1 jusqu'à identifier un obstacle (alors noté u^*) tel que : soit $\texttt{oracle}(u^+) \leq \texttt{BI}$, soit toutes les variables de u^+ sont instanciées. Dans ce dernier cas, si la meilleure solution connue est améliorée, le minorant est mis à jour ($\texttt{BI} = \texttt{oracle}(u^*)$).

(2) Une fois l'obstacle u^* identifié, *la phase de remontée* (ou *waning phase*) cherche une sous–instanciation minimale S de $\{0,1,*\}^n$ telle que $S \sqsubseteq u^*$ et S est un obstacle. S est alors mémorisée sous forme de clause dans la famille $\mathscr{F}$. Chvátal propose d'identifier la clause S en rendant libre les variables précédemment instanciées dans u^+ une à une (sauf la dernière) tant que `oracle`$(S) \leq$ `BI`. La fonction `obstacle` est décrite par l'algorithme 3.4.

Algorithme 3.4 – Fonction obstacle

```
obstacle(u,BI,S)
{
   initialiser une pile vide ;
   for(j = 1,2,...,n)
      if(u_j ≠ *) empiler j ;
   //Phase de descente (waxing phase)
   u+ = u ;
   borne = oracle(u+) ;
   if(borne > BI){
      while(u+ ∉ {0,1}^n) {
         choisir un indice j tel que u+_j = * et une valeur v dans {0,1} ;
         u+_j = v ;
         borne = oracle(u+) ;
         if(borne ≤ BI) break ;
         empiler j ;
      }
      if(borne > BI) BI = borne ;
   }
   //Phase de remontée (waning phase)
   S = u+ ;
   while(la pile n'est pas vide){
      dépiler j ;
      S_j = * ;
      if(oracle(S) > BI) S_j = u+_j ;
   }
   return borne ;
}
```

3.5 Famille path–like

La famille path–like de Resolution search, qui est composée d'un ensemble de clauses $C_1, C_2, ..., C_N$ et d'un ensemble de littéraux $w_1, w_2, ..., w_N$, vérifie les conditions suivantes :

- $w_i \in C_j$ si et seulement si $j = i$, (3.9)
- si $\bar{w}_i \in C_j$ alors $j > i$, (3.10)
- si $w \in C_i$ et $\bar{w} \in C_j$, alors $w = w_i$ ou $\bar{w} = w_j$. (3.11)

Par exemple, l'ensemble de clauses suivant et ses littéraux associés constitue une famille path-like [14] :

$$\begin{aligned}
C_1 &= x_2 x_3 && (w_1 = x_2),\\
C_2 &= x_3 x_6 && (w_2 = x_6),\\
C_3 &= \bar{x}_1 \bar{x}_5 \bar{x}_9 && (w_3 = \bar{x}_1),\\
C_4 &= \bar{x}_2 \bar{x}_5 \bar{x}_8 && (w_4 = \bar{x}_8),\\
C_5 &= x_3 \bar{x}_5 x_7 \bar{x}_9 && (w_5 = x_7),\\
C_6 &= \bar{x}_2 \bar{x}_4 \bar{x}_6 && (w_6 = \bar{x}_4).
\end{aligned}$$

À chaque famille path–like $\mathscr{F}$ est associé un nœud noté $u(\mathscr{F})$ tel que

$$u(\mathscr{F}) = \left(\bigcup_{i=1}^{N} (C_i - w_i) \right) \cup (\bar{w}_1, \bar{w}_2, ..., \bar{w}_N)$$

Le nœud $u(\mathscr{F})$ n'est l'extension d'aucune clause de $\mathscr{F}$. Il correspond au nœud de départ qui est fourni à la fonction `obstacle` pour l'exploration de l'espace. Dans l'exemple précédent, $u(\mathscr{F}) = x_1 \bar{x}_2 x_3 x_4 \bar{x}_5 \bar{x}_6 \bar{x}_7 x_8 \bar{x}_9$.

Dans la phase de descente de la fonction `obstacle`, $u(\mathscr{F})$ est étendu en u^+ de telle manière que $u(\mathscr{F}) \sqsubseteq u^+$ puis, lorsque u^+ devient un obstacle (noté alors u^*), la phase de remontée réduit u^* en S tel que $S \sqsubseteq u^*$. La clause S est donc construite de telle manière qu'il n'existe aucun littéral w tel que $w \in S$ et $\bar{w} \in u(\mathscr{F})$:

$$S_j \neq \bar{u}(\mathscr{F})_j \text{ pour tout } j \in \{1, n\}.$$

Nous considérons les deux cas suivants pour l'ajout d'une nouvelle clause à $\mathscr{F}$: soit il y a eu une phase de descente dans la fonction `obstacle` et $u(\mathscr{F})$ a été étendu en u^+ tel que $S \not\sqsubseteq u(\mathscr{F})$, soit il n'y a pas eu de phase de descente et $u(\mathscr{F})$ a été réduit en S tel que $S \sqsubseteq u(\mathscr{F})$.

3.5.1 Mise à jour après une phase de descente

Dans le cas où certaines variables libres de $u(\mathscr{F})$ ont été fixées à 0 ou 1 dans la fonction obstacle et que $u(\mathscr{F})$ a été étendu en u^+, la clause $u(\mathscr{F})$ n'est pas l'extension de S ($S \not\sqsubseteq u(\mathscr{F})$). S est alors simplement ajoutée à $\mathscr{F}$ et son littéral associé w est choisi parmi $S \backslash u(\mathscr{F})$. L'algorithme 3.5 décrit cette mise à jour, la fonction `mise_a_jour_avec_descente` prend en arguments une famille path-like $\mathscr{F}$ de taille N et une clause $S \not\sqsubseteq u(\mathscr{F})$ telle que $S_j \neq \bar{u}(\mathscr{F})_j$ pour tout $j \in \{1, n\}$:

Algorithme 3.5 – Mise à jour après une phase de descente

```
mise_a_jour_avec_descente(F, S)
{
   Choisir un littéral w tel que w ∈ S\u(F) ;
   C_{N+1} = S ;
   w_{N+1} = w ;
   F = F ∪ C_{N+1} ;
   N = N + 1 ;
}
```

Proposition 3.1 *Soient $\mathscr{F}$ une famille path–like, S une clause telle que $S \not\sqsubseteq u(\mathscr{F})$ et $S_j \neq \bar{u}(\mathscr{F})_j$ pour tout $j \in \{1, n\}$. La mise à jour de $\mathscr{F}$ avec S suivant l'algorithme 3.5 est toujours possible et maintient $\mathscr{F}$ path-like.*

Preuve On note C_{N+1} la clause S ajoutée à $\mathscr{F}$ et w_{N+1} son littéral associé. Par hypothèse, $C_{N+1} \not\sqsubseteq u(\mathscr{F})$ donc il existe un littéral w_{N+1} tel que $w_{N+1} \in S \backslash u(\mathscr{F})$. Comme $w_{N+1} \notin u(\mathscr{F})$, $w_{N+1} \notin \bigcup_{i=1}^{N}(C_i - w_i)$ et comme par hypothèse $S_j \neq \bar{u}(\mathscr{F})_j$ pour tout $j \in \{1, n\}$, $w_{N+1} \notin \{w_1, w_2, \ldots, w_N\}$, on en déduit que $w_{N+1} \notin \bigcup_{i=1}^{N} C_i$ donc la condition (3.9) est respectée. Etant donné que $w_{N+1} \notin \bigcup_{i=1}^{N} C_i$, les conditions (3.10) et (3.11) sont également respectées. Les conditions (3.9), (3.10) et (3.11) sont respectées donc la nouvelle famille est bien path–like. □

3.5.2 Mise à jour sans phase de descente

S'il n'y a pas eu de phase de descente, $\texttt{oracle}(u(\mathscr{F})) \leq \texttt{BI}$ donc $S \sqsubseteq u(\mathscr{F})$. Dans ce cas, chaque littéral de S appartient soit à $\bigcup_{i=1}^{N}(C_i - \{w_i\})$ soit à $\{\bar{w}_1, \bar{w}_2, ..., \bar{w}_N\}$. S peut donc être réduite par résolvantes successives avec toute clause C_i de $\mathscr{F}$ telle que $\bar{w}_i \in S$. Au départ, on définit une clause $R = S \sqsubseteq u(\mathscr{F})$, donc par définition de $u(\mathscr{F})$, $R \sqsubseteq \left(\bigcup_{i=1}^{N}(C_i - w_i)\right) \cup (\bar{w}_1, \bar{w}_2, ..., \bar{w}_N)$. Chaque résolvante de R avec une clause C_i telle que $\bar{w}_i \in R$ réduit R en $R - \{\bar{w}_i\}$ de sorte qu'à la fin nous avons la relation :

$$R \sqsubseteq \bigcup_{i=1}^{N}(C_i - w_i).$$

Si $R = \emptyset$, le nœud racine $(*, *, ..., *)$ est un obstacle. Dans ce cas, le meilleur minorant trouvé (`BI`) est la valeur optimale du problème car aucune extension de $(*, *, ..., *)$ ne mène à une amélioration de `BI`. Si $R \neq \emptyset$, on ajoute R à $\mathscr{F}$ de la manière suivante :

1. trouver le plus petit k tel que $R \subseteq \bigcup_{i=1}^{k}(C_i - w_i)$,
2. choisir un littéral w dans $R - (C_1 \cup C_2 \cup \cdots \cup C_{k-1})$,
3. remplacer C_k par R et poser $w_k = w$,
4. supprimer de $C_{k+1}, C_{k+2}, \ldots, C_N$ toutes les clauses contenant w.

Algorithme 3.6 – Mise à jour sans phase de descente

```
mise_a_jour_sans_descente(F, S)
{
   R = S ;
   for(i = N, N-1, ..., 2, 1)
         if(w̄_i ∈ R) R = R∇C_i ;
   Choisir k, le plus petit indice tel que R ⊆ ⋃_{i=1}^{k}(C_i - w_i) ;
   Choisir w ∈ R - ⋃_{i=1}^{k-1} C_i ;
   C_k = R ;
   w_k = w ;
   for(i = k+1, k+2, ..., N)
      if(w_k ∈ C_i) supprimer C_i de F ;
}
```

Proposition 3.2 *Soient $\mathscr{F}$ une famille path–like et S une clause telle que $S \sqsubseteq u(\mathscr{F})$, alors la mise à jour de $\mathscr{F}$ avec S suivant l'algorithme 3.6 est toujours possible et maintient $\mathscr{F}$ path-like.*

Preuve

Réalisibilité : Nous montrons que l'algorithme 3.6 est réalisable quelque soit $\mathscr{F}$ path-like et $S \sqsubseteq u(\mathscr{F})$. Après la première boucle

```
for(i = N, N-1, ..., 2, 1)
      if(w̄_i ∈ R) R = R∇C_i ;
```

nous avons nécessairement $R \sqsubseteq \bigcup_{i=1}^{N}(C_i - w_i)$, car $R = S \sqsubseteq u(\mathscr{F})$ et toutes les composantes w_i qui sont à $\bar{w}_i$ dans $u(\mathscr{F})$ disparaissent par résolvante avec les clauses C_i. Etant donné que $R \sqsubseteq \bigcup_{i=1}^{N}(C_i - w_i)$, il existe nécessairement un indice $k \in \{1, \ldots, N\}$ dans l'instruction

```
Choisir k, le plus petit indice tel que R ⊆ ⋃_{i=1}^{k}(C_i - w_i) ;
```

car $R \subseteq \bigcup_{i=1}^{k}(C_i - w_i)$. Un simple exemple étant de choisir $k = N$. L'instruction suivante

```
Choisir w ∈ R - ⋃_{i=1}^{k-1} C_i ;
```

consiste à trouver un littéral w dans R qui n'appartient pas à l'ensemble $(C_1 \cup C_2 \cup \cdots \cup C_{k-1})$. Nous pouvons montrer par l'absurde qu'un tel w existe. Soit k le plus petit indice tel que $R \subseteq \bigcup_{i=1}^{k}(C_i - w_i)$. Supposons que tous les littéraux de R soient inclus dans l'ensemble $E = \bigcup_{i=1}^{k-1} C_i$. On peut trivialement restreindre l'ensemble E à $\bigcup_{i=1}^{k-1}(C_i - w_i)$ car $R \subseteq \bigcup_{i=1}^{k}(C_i - w_i)$. On a donc $R \subseteq \bigcup_{i=1}^{k'}(C_i - w_i)$ avec $k' = k - 1$. Il existe donc un indice $k' < k$ tel que $R \subseteq \bigcup_{i=1}^{k'}(C_i - w_i)$ ce qui contredit l'hypothèse de départ. Il existe donc nécessairement un littéral $w \in R - \bigcup_{i=1}^{k-1} C_i$.

$\mathscr{F}$ path-like : Nous montrons que l'algorithme 3.6 maintient la famille $\mathscr{F}$ path–like. Pour cela, nous montrons que les conditions (3.9), (3.10) et (3.11) sont satisfaites. En supposant que $\mathscr{F}$ est path-like lors de l'ajout de C_k, il ne peut exister un littéral w *et* sa négation $\bar{w}$ dans $\bigcup_{i=1}^{N}(C_i - w_i)$ sinon la condition (3.11) n'est pas respectée. Or $C_k \subseteq \bigcup_{i=1}^{N}(C_i - w_i)$ donc il n'existe aucun littéral $w \in C_k$ tel que $\bar{w} \in C_i$ et $\bar{w} \neq w_i$ pour $i \in \{1, \ldots, N\}$, la condition (3.11) est donc toujours respectée. Cette même condition (3.11) certifie que tout littéral $w \neq w_i \in C_i$ pour $i < k$ est différent de $\bar{w}_k$. De plus, $w_k \notin \{w_1, w_2, \ldots, w_{k-1}\}$ car $R \subseteq \bigcup_{i=1}^{N}(C_i - w_i)$ donc il n'existe aucun w_i pour $i < k$ tel que $w_i = \bar{w}_k$, ce qui valide la condition (3.10). Il est possible cependant que la condition (3.9) ne soit pas satisfaite car rien n'interdit qu'il existe un littéral $w_i = w_k$ pour $i > k$. C'est pourquoi on doit supprimer de $C_{k+1}, C_{k+2}, \ldots, C_N$ toutes les clauses qui incluent w_k afin de valider cette dernière condition. □

3.5.3 Mises à jour supplémentaires

Nous avons deux types de mise à jour lorsqu'une clause S est ajoutée à une famille path-like $\mathscr{F}$: soit $S \not\sqsubseteq u(\mathscr{F})$ et la mise à jour se fait suivant l'algorithme 3.5, soit $S \sqsubseteq u(\mathscr{F})$ et la mise à jour se fait suivant l'algorithme 3.6.

Proposition 3.3 *Soient $\mathscr{F}$ une famille path–like et R une clause, alors R peut être ajoutée à $\mathscr{F}$ suivant l'algorithme 3.5 ou 3.6 si et seulement si $R_j \neq \bar{u}(\mathscr{F})_j$ pour tout $j \in \{1, n\}$.*

Preuve Supposons qu'il existe un littéral w tel que $w \in u(\mathscr{F})$ et $\bar{w} \in R$, $S \not\sqsubseteq u(\mathscr{F})$ donc la mise à jour se fait suivant l'algorithme `mise_a_jour_avec_descente`. Nous distinguons deux possibilités : (1) $w \in \bigcup_{i=1}^{N}(C_i - w_i)$, dans ce cas, la condition (3.11) n'est pas respectée car il existe un littéral w tel que $w \in C_i$ et $\bar{w} \in C_{N+1}$ avec $w \neq w_i$ et $w \neq w_{N+1}$; (2) $w \in \bigcup_{i=1}^{N} w_i$, dans ce cas, la condition (3.9) n'est pas respectée car il existe un littéral $w_i \in C_i$ qui appartient également à C_{N+1}.

Supposons qu'il n'existe aucun littéral w tel que $w \in u(\mathscr{F})$ et $\bar{w} \in R$. Nous avons deux possibilités : soit $R \not\sqsubseteq u(\mathscr{F})$ et $R_j \neq \bar{u}(\mathscr{F})_j$ pour tout $j \in \{1, n\}$, donc d'après la proposition 3.1, la mise à jour suivant l'algorithme `mise_a_jour_avec_descente` est possible ; soit $R \sqsubseteq u(\mathscr{F})$ et d'après la proposition 3.2, la mise à jour suivant l'algorithme `mise_a_jour_sans_descente` est possible. □

D'après la proposition 3.3, si après une mise à jour par la procédure `mise_a_jour_sans_descente`, aucun littéral de $u(\mathscr{F})$ n'a sa négation dans S alors il est possible d'ajouter une nouvelle fois S à $\mathscr{F}$. Chvátal exprime cette condition d'une manière moins intuitive mais plus efficace d'un point de vue algorithmique. Après la mise à jour, la clause C_k de $\mathscr{F}$ a été remplacée par R et un nouveau littéral $w_k \in C_k$ a été choisi. Il propose alors simplement de vérifier si le littéral w_k n'appartient pas à S auquel cas $\mathscr{F}$ peut être à nouveau mise à jour en ajoutant S. Cette condition est en fait une manière de vérifier si la proposition 3.3 est valide. Pour toutes les clauses autres que C_k, on a $S_j \neq \bar{u}(\mathscr{F})_j$ pour $j = 1, ..., n$ car ces clauses appartenaient à $\mathscr{F}$ avant la mise à jour et les seuls littéraux w_i tels que $w_i \in C_i$ et $\bar{w}_i \in S$ ont été éliminés de S par résolvante. Pour C_k, cette condition n'est valide que si $w_k \notin S$ car si $w_k \in S$, par construction de $u(\mathscr{F})$, $\bar{w}_k \in u(\mathscr{F})$ et selon la proposition 3.3, S ne peut pas être ajoutée à $\mathscr{F}$. L'algorithme 3.7 décrit la mise à jour dans le cas général.

Algorithme 3.7 – Mise à jour

```
mise_a_jour(F, S)
{
   do{
      if(S ⋢ u(F))
         mise_a_jour_avec_descente(F, S) ;
      else if(S ⊑ u(F))
         mise_a_jour_sans_descente(F, S) ;
   } while(u(F) ≠ S̄_i pour i = 1, ..., n) ;
}
```

3.5.4 Choix des littéraux associés aux clauses

Resolution search offre la possibilité d'implémenter deux politiques de branchement distinctes : (i) une politique d'*affectation* et (ii) une politique de *remise en cause*. La politique d'affectation est la stratégie de branchement de la fonction `obstacle`, elle consiste à déterminer quelle variable libre sera instanciée et à quelle valeur elle sera instanciée dans la phase de descente, ce qui correspond à l'instruction

`choisir un indice` j `tel que` $u_j^+ = *$ `et une valeur` v `dans` $\{0, 1\}$.

La deuxième politique est celle qui consiste à choisir, lorsqu'un obstacle est identifié, quelle instanciation de cet obstacle sera fixée à sa valeur opposée dans les prochaines itérations, ce qui correspond aux instructions

`Choisir un littéral` w `tel que` $w \in S \backslash u(\mathscr{F})$ et `Choisir` $w \in R - \bigcup_{i=1}^{k-1} C_i$

dans les algorithmes `mise_a_jour_avec_descente` et `mise_a_jour_sans_descente`.

La première politique correspond aux stratégies de branchement classiques que l'on trouve dans la plupart des méthodes d'énumération. La deuxième politique n'est pas si

courante. Dans la plupart des algorithmes de résolution, le choix de la variable sur laquelle est effectué le retour en arrière est lié à l'ordre chronologique. Dans le cas du backtracking ou de l'énumération implicite, il s'agit systématiquement de la dernière. Dans le cas de `DDB` et de `DBT`, il ne s'agit pas nécessairement de la dernière variable instanciée mais de la dernière variable appartenant à l'obstacle identifié [35, 2]. Cette caractéristique permet une exploration plus souple de l'espace de recherche. La structure path–like impose certaines règles dans le choix de w mais une certaine liberté est tout de même donnée par rapport à ces différentes méthodes.

3.5.5 Intérêt de la structure path–like pour la simplification des clauses

Considérons la *force* d'une famille path-like $\mathscr{F}$ comme étant le nombre de vecteurs qu'elle recouvre, c'est–à–dire le nombre de vecteurs $x \in \{0,1\}^n$ qui satisfont $C \sqsubseteq x$ pour au moins une clause C de $\mathscr{F}$. On en déduit que moins les clauses de $\mathscr{F}$ contiennent de littéraux, plus sa force est haute. Il est donc intéressant, du point de vue de la réduction de l'espace de recherche, de simplifier au maximum les clauses de manière à maximiser la force $\mathscr{F}$. La structure path-like fait en sorte que ni w_i ni $\bar{w}_i$ n'appartiennent à des clauses C_j pour $j < i$ et w_i n'appartient pas à des clauses C_j pour $j > i$. De cette manière, lorsque le littéral w_i, qui est à $\bar{w}_i$ dans R, est supprimé de R par résolvante entre C_i et R, il n'apparaîtra plus dans R dans la suite de la mise à jour. Reprenons l'exemple précédent

$$\begin{aligned}
C_1 &= x_2x_3 && (w_1 = x_2),\\
C_2 &= x_3x_6 && (w_2 = x_6),\\
C_3 &= \bar{x}_1\bar{x}_5\bar{x}_9 && (w_3 = \bar{x}_1),\\
C_4 &= \bar{x}_2\bar{x}_5\bar{x}_8 && (w_4 = \bar{x}_8),\\
C_5 &= x_3\bar{x}_5x_7\bar{x}_9 && (w_5 = x_7),\\
C_6 &= \bar{x}_2\bar{x}_4\bar{x}_6 && (w_6 = \bar{x}_4).
\end{aligned}$$

Supposons que l'on ajoute $S = \bar{x}_5\bar{x}_6\bar{x}_7$, la mise à jour effectue les simplifications suivantes

$$\begin{aligned}
R &= S\nabla C_5 = x_3\bar{x}_5\bar{x}_6\bar{x}_9\\
R &= R\nabla C_2 = x_3\bar{x}_5\bar{x}_9
\end{aligned}$$

Supposons que l'on ne respecte pas les conditions path-like et que $C_2 = x_3x_6x_7$. Lors de la première simplification

$$R = S\nabla C_5 = x_3\bar{x}_5\bar{x}_6\bar{x}_9$$

le littéral x_7 disparaît de R, mais lors de la deuxième simplification

$$R = R\nabla C_2 = x_3\bar{x}_5x_7\bar{x}_9$$

x_7 est réintroduit dans R. La structure path–like permet ainsi d'assurer qu'une fois le littéral w_i éliminé par résolvante avec R, il ne réapparaîtra pas dans R. De cette manière, on assure une simplification efficace des clauses ajoutées à $\mathscr{F}$.

3.6 Convergence finie

Nous montrons dans cette section que pour la résolution d'un problème d'optimisation à variables binaires, Resolution search converge vers une solution optimale en un nombre fini d'itérations.

3.6.1 Preuve d'optimalité

Resolution search est basée sur le principe de preuve par résolution et réfutation [60]. Nous allons voir que la suite de clauses générées chronologiquement durant le processus de recherche représente un certificat d'optimalité.

Soit r une valeur réelle, on dit qu'une clause C est *r-forçante* si pour tout $x \in \{0,1\}^n$ tels que $C \sqsubseteq x$, on a $\texttt{oracle}(x) \leq r$.

Proposition 3.4 *Soit C_1 et C_2 deux clauses r–forçantes, alors $C_3 = C_1 \nabla C_2$ est également r–forçante.*

Preuve Nous démontrons la proposition en raisonnant par l'absurde. Supposons que C_1, C_2 soient deux clauses r–forçantes et que $C_3 = C_1 \nabla C_2$ ne soit pas r-forçante, c'est–à–dire qu'il existe u tel que $C_3 \sqsubseteq u$ et $\texttt{oracle}(u) > r$. Si $\texttt{oracle}(u) > r$, alors u n'est l'extension d'aucune des clauses C_1, C_2, c'est–à–dire que $C_1 \not\sqsubseteq u$ et $C_2 \not\sqsubseteq u$. Par extension, $(C_1 - \{w\}) \not\sqsubseteq u$ et $(C_2 - \{\bar{w}\}) \not\sqsubseteq u$ et donc $C_3 = C_1 \nabla C_2 \not\sqsubseteq u$ ce qui contredit l'hypothèse de départ. □

Proposition 3.5 *Une preuve par résolution et réfutation que r est la valeur optimale d'un problème de maximisation P, c'est-à-dire qu'il n'existe aucun vecteur u dans $\{0,1\}^n$ tel que* `oracle(u) > r`*, consiste en une suite de clauses $R_1, \ldots, R_N$ telle que*
(i) pour tout $k = 1, 2, \ldots, N$, $\texttt{oracle}(R_k) \leq r$ ou $R_k = R_i \nabla R_j$ pour i, $j < k$,
(ii) $R_N = \emptyset$.

Preuve (1) On prouve d'abord par récurrence que la construction d'une famille $R_1, \ldots, R_N$ suivant la condition (i) implique que $\texttt{oracle}(R_k) \leq r$ pour $k = 1, \ldots, N$. La propriété est vraie aux rangs 1 et 2 car dans les deux cas, il ne peut exister deux indices $i, j < k$. Supposons maintenant que la propriété soit vraie au rang n, alors $\texttt{oracle}(R_k) \leq r$ pour $k = 1, \ldots, n$. Au rang $n+1$, nous avons donc : soit $\texttt{oracle}(R_{n+1}) \leq r$, ce qui valide la propriété, soit $R_{n+1} = R_i \nabla R_j$ pour $i, j < n+1$. Or, d'après la proposition 3.4, si R_i et R_j

sont r-forçantes, alors $R_i \nabla R_j$ est r–forçante. Donc $\texttt{oracle}(R_{n+1}) \leq r$ et la propriété est vraie au rang $n+1$.

(2) Nous montrons maintenant par l'absurde qu'une famille $R_1, R_2, \ldots, R_N$ satisfaisant (i) et (ii) est une preuve que r est la valeur optimale du problème P. Supposons qu'il existe un vecteur $u \in \{0,1\}^n$ tel que $\texttt{oracle}(u) > r$. D'après (i), $\texttt{oracle}(R_k) \leq r$ pour $k = 1, \ldots, N$, donc $R_k \not\sqsubseteq u$ pour $k = 1, ..., N$. D'après (ii), $R_N = \emptyset$, donc $\emptyset \not\sqsubseteq u$, ce qui contredit l'hypothèse de départ. □

Supposons que la suite de clauses générées chronologiquement par Resolution search soit mémorisée dans une famille notée $\mathcal{F}$. Le déroulement de Resolution search nous amène à effectuer les opérations suivantes :

- Si $\texttt{oracle}(u) > \texttt{BI}$, alors $\texttt{BI} = \texttt{oracle}(u)$.
- Si S est une clause telle que $\texttt{oracle}(S) \leq \texttt{BI}$, alors S peut être ajoutée à $\mathcal{F}$.
- Si R et C sont en conflit dans $\mathcal{F}$, alors $R \nabla C$ peut être ajoutée à $\mathcal{F}$.

À la fin de la procédure, la clause $S = \emptyset$ est ajoutée à $\mathcal{F}$. Les conditions (i) et (ii) de la proposition 3.5 sont donc respectées et $\mathcal{F}$ représente une preuve que BI est la valeur optimale du problème. Finalement, la famille path–like $\mathscr{F}$ correspond à un sous–ensemble de la famille $\mathcal{F}$, certaines clauses ayant été supprimées pendant les différentes mises à jour.

3.6.2 Convergence

Nous montrons que Resolution search converge vers l'optimum en un nombre fini d'itérations. Dans le cas de la résolution d'un problème à n variables, on peut définir la force d'une famille de clauses $\mathscr{F}$ notée $\tau(\mathscr{F})$ égale à 2^{-n} fois le nombre de vecteurs $x \in \{0,1\}^n$ étant l'extension d'au moins une clause de $\mathscr{F}$. Cette valeur comprise entre 0 et 1 représente le ratio du nombre de solutions coupées de l'espace de recherche par $\mathscr{F}$ sur le nombre total de solutions. Comme le précise Chvátal, $\tau(\mathscr{F})$ n'augmente pas systématiquement à chaque itération de Resolution search. Cependant, on peut identifier une borne inférieure de la force de la famille qui augmente à chaque itération. On note n_i le nombre de variables couvertes par $C_1, C_2, \ldots, C_i$ telle que

$$n_i = |\bigcup_{k=1}^{i} C_k|$$

Cette borne inférieure $\sigma(\mathscr{F})$ de $\tau(\mathscr{F})$ pour une famille $(C_1, C_2, \ldots, C_N)$ est définie de la manière suivante :

$$\sigma(\mathscr{F}) = \sum_{i=1}^{N} 2^{-n_i}.$$

Dans la suite, on considère $\sigma(\mathscr{F})$ comme la force de la famille $\mathscr{F}$.

Théorème 1 *Lors de la résolution d'un problème à n variables par Resolution search, le nombre d'appels à la fonction* `obstacle` *est au plus égal à* 2^n.

Preuve Lors de la résolution d'un problème à n variables, toute famille path–like non vide respecte l'inégalité suivante :

$$1 \leq n_1 < n_2 < \cdots < n_N \leq n \tag{3.12}$$

sa force, étant définie comme $\sigma(\mathscr{F}) = 2^{-n_1} + 2^{-n_2} + \cdots + 2^{-n_N}$ avec $0 < n_i \leq n$ pour $i \in \{1, N\}$, sa valeur est un multiple positif de 2^{-n}. On a donc

$$0 < \sigma(\mathscr{F}) = \lambda \cdot 2^{-n} \leq 1, \quad \lambda \in \mathbb{N} \tag{3.13}$$

Nous montrons à présent que $\sigma(\mathscr{F})$ croît strictement à chaque appel de la fonction `obstacle`. Lors de l'ajout d'une clause S à $\mathscr{F}$, nous distinguons les deux cas de figure :

1. $S \sqsubseteq u(\mathscr{F})$, dans ce cas S est simplement ajoutée à $\mathscr{F}$ donc la force augmente trivialement.
2. $S \not\sqsubseteq u(\mathscr{F})$, dans ce cas, $(C_1, C_2, \ldots, C_N)$ est étendue en $(C_1, C_2, \ldots, C_{k-1}, R, \ldots, C_N)$ avec éventuellement des suppressions de clauses entre C_{k+1} et C_N. Puisque la clause ajoutée R est telle que $R \subseteq \bigcup_{i=1}^{k}(C_i - \{w_i\})$, toutes les variables couvertes par $C_1, C_2, \ldots, C_{k-1}, R$ sont couvertes par $C_1, C_2, \ldots, C_k$ et, de plus, w_k n'est pas couverte par $C_1, C_2, \ldots, C_{k-1}$. On a donc l'inégalité

   $$|C_1 \cup C_2 \cup \cdots \cup C_{k-1} \cup R| < |C_1 \cup C_2 \cup \cdots \cup C_k|$$

   et donc,

   $$\sigma(C_1, C_2, \ldots, C_{k-1}, R) > \sigma(C_1, C_2, \ldots, C_k)$$

   De plus, certaines clauses de $C_{k+1}, ..., C_N$ ont pu être supprimées durant la mise à jour, donc nécessairement, la force de la nouvelle famille est strictement supérieure à la force de l'ancienne famille. □

3.7 Exemple

Dans cette section, nous donnons un exemple d'exécution de Resolution search sur le MKP suivant :

$$\begin{array}{lrcrcrcrcrcl}
(P^c) & 10x_1 & + & 10x_2 & + & 8x_3 & + & 8x_4 & + & 7x_5 & & \\
\text{sujet à} & 5x_1 & + & 2x_2 & + & 3x_3 & + & 1x_4 & + & 3x_5 & \leq & 10 \\
& 2x_1 & + & 4x_2 & + & 2x_3 & + & 5x_4 & + & x_5 & \leq & 11 \\
& x = {}^T(x_0, x_1, x_2, x_3, x_4) \in \{0,1\}^5 & & & & & & & & & &
\end{array}$$

Nous détaillons le déroulement de Resolution search pour la résolution du problème P^c. Nous illustrons dans le tableau 3.1 les 17 itérations nécessaires à la résolution de ce problème. Nous verrons ensuite le détail de certaines des itérations. Dans cet exemple, la stratégie de branchement consiste à affecter la valeur 0 à la première variable libre dans l'ordre lexicographique, et la stratégie de remise en cause consiste à choisir le littéral $w \in R - \bigcup_{i=1}^{k-1} C_i$ également dans l'ordre lexicographique. La colonne *Itération* est le numéro de l'itération, la colonne $u(\mathscr{F})$ représente le vecteur $u(\mathscr{F})$ à l'itération courante, la colonne *Explication* explique le déroulement de la fonction `obstacle` (solution trouvée ou échec rencontré), $\mathscr{F}$ représente la famille path–like courante (sans les littéraux w_i associés aux clauses) et $\sigma(\mathscr{F})$ donne la force de la famille courante.

Tab. 3.1 – Déroulement de Resolution search pour résoudre le problème P^c

Itération	$u(\mathscr{F})$	S	Explication	$\mathscr{F}$	$\sigma(\mathscr{F})$
1	$(*,*,*,*,*,)$	$x_1x_2x_3x_4x_5$	`oracle =0.00 < 0`	$\{x_1x_2x_3x_4x_5\}$	0.03125
2	$(1,0,0,0,0,)$	$\bar{x}_1x_2x_3x_4x_5$	`sol. 10`	$\{x_2x_3x_4x_5\}$	0.06250
3	$(*,1,0,0,0,)$	$x_1x_3x_4x_5$	`oracle =10.00 < 10`	$\{x_2x_3x_4x_5, x_1x_3x_4x_5\}$	0.09375
4	$(1,1,0,0,0,)$	$\bar{x}_1\bar{x}_2x_3x_4x_5$	`sol. 20`	$\{x_3x_4x_5\}$	0.12500
5	$(*,*,1,0,0,)$	$x_1x_4x_5$	`oracle =18.00 < 20`	$\{x_3x_4x_5, x_1x_4x_5\}$	0.18750
6	$(1,*,1,0,0,)$	$x_2x_4x_5$	`oracle =18.00 < 20`	$\{x_3x_4x_5, x_1x_4x_5, x_2x_4x_5\}$	0.21875
7	$(1,1,1,0,0,)$	$\bar{x}_1\bar{x}_2\bar{x}_3x_4x_5$	`sol. 28`	$\{x_4x_5\}$	0.25000
8	$(*,*,*,1,0,)$	x_1x_5	`oracle =26.00 < 28`	$\{x_4x_5, x_1x_5\}$	0.37500
9	$(1,*,*,1,0,)$	x_2x_5	`oracle =26.00 < 28`	$\{x_4x_5, x_1x_5, x_2x_5\}$	0.43750
10	$(1,1,*,1,0,)$	$\bar{x}_1\bar{x}_2\bar{x}_4$	`oracle =` $-\infty$	$\{x_5, \bar{x}_1\bar{x}_2\bar{x}_4\}$	0.56250
11	$(0,1,*,1,1,)$	x_1x_3	`oracle =25.00 < 28`	$\{x_5, \bar{x}_1\bar{x}_2\bar{x}_4, x_1x_3\}$	0.59375
12	$(0,1,1,1,1,)$	$\bar{x}_2\bar{x}_3\bar{x}_4$	`oracle =` $-\infty$	$\{x_5, \bar{x}_2\bar{x}_4, x_1x_3\}$	0.65625
13	$(0,0,1,1,1,)$	x_1x_2	`oracle =23.00 < 28`	$\{x_5, \bar{x}_2\bar{x}_4, x_1\bar{x}_4\}$	0.68750
14	$(1,0,*,1,1,)$	$\bar{x}_1x_2\bar{x}_5$	`oracle =27.67 < 28`	$\{x_5, \bar{x}_4, \bar{x}_1x_2\bar{x}_5\}$	0.81250
15	$(0,0,*,0,1,)$	x_1x_2	`oracle =15.00 < 28`	$\{x_5, \bar{x}_4, x_2\}$	0.87500
16	$(*,1,*,0,1,)$	x_1x_4	`oracle =25.00 < 28`	$\{x_5, \bar{x}_4, x_2, x_1x_4\}$	0.93750
17	$(1,1,*,0,1,)$	$\bar{x}_1x_4$	*optimum.*	$\{\emptyset\}$	1.00000

À l'itération 1, l'algorithme considère le vecteur $u = (*,*,*,*,*)$. La fonction `obstacle` choisit les variables libres dans l'ordre lexicographique et leur affecte la valeur 0 jusqu'à obtenir l'obstacle $u^* = (0,0,0,0,0)$. On obtient donc le premier minorant $\texttt{BI} = \texttt{oracle}(u^*) = 0$. L'obstacle $S = x_1x_2x_3x_4x_5$ est alors ajouté à la famille path–like et on choisit le littéral $w_1 \in S$ dans l'ordre lexicographique, c'est–à–dire x_1. À ce stade, la famille path–like correspond à

$$\mathscr{F} := \quad C_1 = x_1x_2x_3x_4x_5 \quad (w_1 = x_1),$$

Le nœud de départ associé à $\mathscr{F}$ est $u(\mathscr{F}) = (1,0,0,0,0)$. À l'itération 2, on a directement $\texttt{BI} = \texttt{oracle}(u(\mathscr{F})) = 10$ qui devient la nouvelle meilleure borne inférieure connue. On ajoute alors l'obstacle $S = \bar{x}_1x_2x_3x_4x_5$ à $\mathscr{F}$. Comme $S \sqsubseteq u(\mathscr{F})$, une simplification est possible, on rentre dans le cas de l'algorithme `mise_a_jour_sans_descente`. On a $R = S \nabla C_1 = x_2x_3x_4x_5$ et comme $R \sqsubseteq C_1 - w_1$, on remplace C_1 par R. On choisit alors $w_1 = x_2$

et la famille $\mathscr{F}$ devient

$$\mathscr{F} := \quad C_1 = x_2x_3x_4x_5 \quad (w_1 = x_2),$$

Dans l'itération 3, on étend le nœud $u(\mathscr{F}) = (*, 1, 0, 0, 0)$ en $u^* = (0, 0, 0, 1, 0)$. $\texttt{oracle}(u^*) = 10.00 \leq 10$ et on déduit l'obstacle $S = x_1x_3x_4x_5$. Comme $S \not\sqsubseteq u(\mathscr{F})$, on ajoute simplement S à $\mathscr{F}$ qui devient :

$$\begin{aligned}\mathscr{F} := \quad & C_1 = x_2x_3x_4x_5 \quad (w_1 = x_2),\\ & C_2 = x_1x_3x_4x_5 \quad (w_2 = x_1),\end{aligned}$$

À l'itération 4, $u(\mathscr{F}) = \bar{x}_2x_3x_4x_5 \cup \bar{x}_1x_3x_4x_5$ ce qui revient, sous forme de vecteur, à $u(\mathscr{F}) = (1, 1, 0, 0, 0)$. On a alors une mise à jour de la borne inférieure $\texttt{BI} = \texttt{oracle}(u(\mathscr{F})) = 20$. On ajoute alors l'obstacle $S = \bar{x}_1\bar{x}_2x_3x_4x_5$ et comme $S \sqsubseteq u(\mathscr{F})$ certaines simplifications sont possibles. On a tout d'abord $R = S\nabla C_2 = \bar{x}_2x_3x_4x_5$, puis $R = R\nabla C_1 = x_3x_4x_5$. Comme $R \sqsubseteq C_1 - w_1$, on remplace C_1 par R et on choisit $w_1 = x_3$. Puisque $x_3 \in C_2$, on supprime C_2 de $\mathscr{F}$. Après la mise à jour, $\mathscr{F}$ devient :

$$\mathscr{F} := \quad C_1 = x_3x_4x_5 \quad (w_1 = x_3),$$

Par le même procédé, on poursuit jusqu'à l'itération 17 dans laquelle

$$\begin{aligned}\mathscr{F} := \quad & C_1 = x_5 \quad & (w_1 = x_5),\\ & C_2 = \bar{x}_4 \quad & (w_2 = \bar{x}_4),\\ & C_3 = x_2 \quad & (w_3 = x_2),\\ & C_4 = x_1x_4 \quad & (w_4 = x_1),\end{aligned}$$

On a alors $u(\mathscr{F}) = (1, 1, *, 0, 1)$ et après la phase de descente, $\texttt{oracle}(1, 1, 0, 0, 1) = 27 \leq 28$. La phase de remontée déduit l'obstacle $S = \bar{x}_1\bar{x}_2\bar{x}_5$. $S \sqsubseteq u(\mathscr{F})$ donc des simplifications sont possibles : on a $R = S\nabla C_4 = \bar{x}_2x_4\bar{x}_5$, $R = R\nabla C_3 = x_4\bar{x}_5$ puis $R = R\nabla C_2 = \bar{x}_5$ et enfin $R = R\nabla C_1 = \emptyset$. À la fin de cette mise à jour, on a donc $R = \emptyset$ donc 28 est la valeur optimale de P^c.

Cet exemple illustre le fait que l'on retrouve le phénomène de trashing dans Resolution search. Lors de l'itération 13, la procédure déduit que $S = x_1x_2$ est un obstacle et redécouvre cette information à l'itération 15. Ce phénomène est dû à la suppression des obstacles pendant le processus de mise à jour de la famille path–like. C'est en quelque sorte le *prix à payer* pour conserver $\mathscr{F}$ path–like et maintenir une complexité spatiale polynomiale.

3.8 Remarque sur l'implémentation des clauses

Nous proposons une implémentation des clauses qui permet une gestion efficace des différents opérateurs de résolvante et d'union.

Une clause C peut être représentée par deux vecteurs booléens : le premier noté `v` contient les valeurs des littéraux et le deuxième noté `m` est un masque tel que `m[i]` $= 1$ si $C_i \neq *$ et 0 sinon. Une clause C_i est représentée par ses deux vecteurs $\mathtt{v}_i$ et $\mathtt{m}_i$. Nous présentons les tables de vérité des deux opérateur pour `v` et `m`. L'opérateur de résolvante ($C_3 = C_1 \nabla C_2$) est représenté en figure 3.1 et l'opérateur d'union ($C_3 = C_1 \cup C_2$) en figure 3.2. Ces différentes opérations logiques peuvent aisément être traitées grâce au conteneur `bitset`[4] disponible dans les librairies standards des langages `C` et `C++`. Ce conteneur permet, de plus, un accès en temps constant à chaque bit. Ainsi, des opérations entre clauses comme la résolvante ou l'union peuvent être réalisées beaucoup plus rapidement qu'avec une structure de type tableau et des boucles « `for` ».

Fig. 3.1 – Tables de vérité pour l'opérateur de résolvante

$\mathtt{v}_1\mathtt{m}_1$ \ $\mathtt{v}_2\mathtt{m}_2$	00	01	11	10
00	X	0	1	X
01	0	0	X	0
11	1	X	1	1
10	X	0	1	X

$\mathtt{v}_3 = \mathtt{v}_1 \cdot \mathtt{m}_1 + \mathtt{v}_2 \cdot \mathtt{m}_2$

$\mathtt{v}_1\mathtt{m}_1$ \ $\mathtt{v}_2\mathtt{m}_2$	00	01	11	10
00	0	1	1	0
01	1	1	0	1
11	1	0	1	1
10	0	1	1	0

$\mathtt{m}_3 = (\mathtt{m}_1 \oplus \mathtt{m}_2) + \mathtt{m}_1 \cdot \mathtt{m}_2 \cdot (\overline{\mathtt{v}_1 \oplus \mathtt{v}_2})$

Fig. 3.2 – Tables de vérité pour l'opérateur d'union

$\mathtt{v}_1\mathtt{m}_1$ \ $\mathtt{v}_2\mathtt{m}_2$	00	01	11	10
00	X	0	1	X
01	0	0	X	0
11	1	X	1	1
10	X	0	1	X

$\mathtt{v}_3 = \mathtt{v}_1 \cdot \mathtt{m}_1 + \mathtt{v}_2 \cdot \mathtt{m}_2$

$\mathtt{v}_1\mathtt{m}_1$ \ $\mathtt{v}_2\mathtt{m}_2$	00	01	11	10
00	0	1	1	0
01	1	1	1	1
11	1	1	1	1
10	0	1	1	0

$\mathtt{m}_3 = \mathtt{m}_1 + \mathtt{m}_2$

3.9 Expériences numériques

Dans cette section, nous exposons les résultats de plusieurs expériences numériques réalisées sur des instances de MKP. En premier lieu, nous mettons en avant l'impact négatif qu'a la phase de remontée sur la vitesse d'exécution de Resolution search dans le cas de l'implémentation originale proposée par Chvátal. Nous proposons ensuite une comparaison entre l'énumération implicite et Resolution search sur la résolution des instances cb5.250 de la `OR-Library`.

[4] `http ://www.sgi.com/tech/stl/bitset.html`.

3.9.1 Impact négatif de la phase de remontée

Dans le cas d'une stricte implémentation de l'algorithme tel que décrit dans [14], `oracle` est considérée comme une boîte noire qui résout le programme linéaire $\bar{P}(u)$ pour une affectation partielle u. À chaque phase de remontée, on peut alors exécuter jusqu'à $n-1$ appels à l'algorithme du simplexe pour évaluer l'affectation partielle $S \sqsubseteq u^*$. Les premières expérimentations réalisées avec la phase de remontée ont montré des temps de calcul relativement longs. Le temps de calcul consommé à générer l'obstacle S de cette manière, ne vaut pas le temps gagné par la réduction de l'espace de recherche qui en résulte. Afin de donner un bref aperçu du comportement de Resolution search avec et sans phase de remontée, nous montrons dans le tableau 3.2 les résultats obtenus sur 30 instances de MKP ayant 25 variables et 5 contraintes. Dans ce tableau, # `oracle` est le nombre d'appels à `oracle` pour résoudre l'ensemble des 30 instances et t(s) est la somme des temps en seconde.

TAB. 3.2 – Impact négatif de la phase de remontée

avec phase de remontée		sans phase de remontée	
# `oracle`	t (s)	# `oracle`	t (s)
6018059	22.9	**328898**	**3.2**

Chvátal explique qu'il est possible d'améliorer la phase de remontée en ne considérant pas `oracle` comme une boîte noire. Il est en effet possible, par l'intermédiaire des coûts réduits, de générer des obstacles sans faire appel à `oracle`. Nous montrons, dans les sections suivantes, que cela permet une nette amélioration par rapport à la version originale. Pour l'instant, nous nous contentons d'une version de Resolution search sans phase de remontée.

3.9.2 Stratégies de branchement

Nous avons mené une première expérimentation afin de vérifier l'influence des stratégies de branchement sur Resolution search. Nous avons comparé les résultats donnés par Resolution search sans phase de remontée avec une énumération implicite pour la résolution des instances cb5.100 de la `OR-Library`. Dans le but d'accélérer le temps de résolution, nous avons utilisé la décomposition de l'espace de recherche en hyperplans ainsi que la prise en compte de la contrainte des coûts réduits (voir chapitre 2). À la différence de l'algorithme du chapitre 2, nous ne donnons pas de minorant de départ. Rappelons que la prise en compte des coûts réduits permet de générer la contrainte suivante :

$$\sum_{j \in N^-} |\bar{c}_j| x_j + \sum_{j \in N^+} |\bar{c}_j|(1 - x_j) \leq \lfloor \mathtt{BS} \rfloor - \mathtt{BI} - 1$$

où $\bar{c}$ est le vecteur des coûts réduits associés à la solution optimale de la relaxation continue du problème de valeur `BS`. N^- est l'ensemble des indices des variables hors base à leur borne inférieure et N^+ celui des variables hors base à leur borne supérieure.

L'implémentation de Resolution search et de l'énumération implicite utilise ces deux résultats. Chaque sous–problème $\bar{P}(k)$ est résolu au départ afin de générer la contrainte des coûts réduits correspondante. Les différents sous–problèmes $P(k)$ pour $k = k_{min}, ..., k_{max}$ sont alors résolus l'un après l'autre dans l'ordre décroissant des bornes $v(\bar{P}(k))$. On suppose que la fonction `oracle(`u`)` intègre la vérification de la validité de l'affectation u avec la contrainte des coûts réduits : si $\sum_{j\in\Omega} \bar{c}_j > gap$ avec Ω l'ensemble des indices j tels que $u_j = 1 - \bar{x}_j$ et x_j est hors base, alors `oracle`$(u) = -\infty$.

Nous proposons deux stratégies de branchement : la première stratégie notée *conv.* consiste à brancher sur la variable libre x_j de plus grand coût réduit et à la fixer à la valeur entière la plus proche de sa valeur optimale $\bar{x}_j$; la deuxième stratégie notée *lexico.* consiste à choisir la première variable libre dans l'ordre lexicographique et à la fixer également à la valeur entière la plus proche de sa valeur optimale.

Dans le cas de Resolution search, la politique de remise en cause consiste à choisir la variable la plus récemment instanciée. Les résultats obtenus sont exposés dans le tableau 3.3. La colonne *opt.* donne le nombre moyen d'appels à `oracle` nécessaire à l'obtention de la valeur optimale et la colonne *preuve* donne le nombre moyen d'appels à `oracle` nécessaire à la preuve d'optimalité.

Tab. 3.3 – Influence des stratégies de branchement pour Resolution search

Instance	énumération implicite				Resolution search			
	conv.		lexico.		conv.		lexico.	
	opt.	preuve	opt.	preuve	opt.	preuve	opt.	preuve
0-9	110610	211458	267381	369750	110649	211505	267052	369410
10-19	97016	200169	228398	355696	97046	200201	228101	339970
20-29	24566	70101	50957	88754	24565	70098	50863	88658

Le nombre de nœuds requis par les deux méthodes pour chaque stratégie de branchement est sensiblement le même. Telle que nous l'avons implémentée, c'est–à–dire sans phase de remontée et avec la même politique de branchement, Resolution search se comporte de manière similaire à l'énumération implicite. Le nombre de nœuds est sensiblement égal et les stratégies de branchement affectent tout autant les deux méthodes. L'intérêt de Resolution search réside dans la souplesse qu'elle offre à l'exploration de l'espace de recherche. En effet, jusqu'ici, nous n'avons exploité ni un critère de remise en cause ni l'identification d'obstacles minimaux.

3.9.3 Critère de remise en cause

Nous proposons une deuxième expérimentation dans laquelle la stratégie de remise en cause de Resolution search est modifiée. Au lieu de choisir la dernière variable instanciée, nous choisissons la variable de plus grand coût réduit dans S. Pour cette expérimentation, nous avons choisi la politique *lexico.*

TAB. 3.4 – Influence du critère de remise en cause

Instance	ordre chronologique		plus grand coût réduit	
	opt.	preuve	opt.	preuve
0-9	267052	369410	**72669**	**162704**
10-19	228101	339970	**73970**	**164237**
20-29	50863	88658	**17621**	**48611**

Cet exemple montre que le critère de remise en cause peut avoir un impact important sur l'efficacité de Resolution search. On constate, dans ce cas, une nette diminution du nombre moyen d'appels à `oracle` pour l'obtention de la solution optimale et pour la preuve d'optimalité. Dans cette expérimentation, 17 valeurs optimales sur 30 ont été trouvées avec un nombre de nœuds inférieur et 19 preuves d'optimalité sur 30 ont été obtenues avec un nombre de nœuds inférieur qu'avec la remise en cause suivant l'ordre chronologique.

3.9.4 Identification des obstacles

Nous proposons une première méthode permettant d'identifier des obstacles sans procéder à une phase de remontée. Elle consiste, dans le cas où l'échec est provoqué par la violation de la contrainte des coûts réduits, à ne conserver dans S que les instanciations des variables fixées à l'opposé de leur valeur optimale. En effet, nous avons vu dans le chapitre 2 que la contrainte des coûts réduits peut être exprimée de la manière suivante :

$$\sum_{j \in \Omega(u)} \bar{c}_j \leq \lfloor \mathtt{BS} \rfloor - \mathtt{BI} - 1$$

avec $\Omega(u)$ l'ensemble des indices des variables x_j instanciées à l'opposé de leur valeur optimale (1 - $\bar{x}_j$) dans l'instanciation partielle u. Si cette contrainte est violée, l'obstacle S contient uniquement les instanciations des variables de $\Omega(u)$ au lieu de considérer toutes les instanciations de u. Le tableau 3.5 donne le nombre moyen d'appels à `oracle` nécessaires à l'obtention de la solution optimale et de la preuve d'optimalité dans le cas $S = u$ et dans le cas $S = \Omega(u)$.

TAB. 3.5 – Influence de l'identification d'obstacles

Instance	$S = u$		$S = \Omega(u)$	
	opt.	preuve	opt.	preuve
0-9	72669	162704	**69286**	**145395**
10-19	73970	164237	**71286**	**155152**
20-29	17621	48611	**17414**	**45696**

On constate de nouveau une amélioration par rapport à une version de Resolution search sans phase de remontée.

3.9.5 Comparaison avec l'énumération implicite

Pour évaluer l'efficacité de Resolution search en termes de calcul de minorants et en termes de temps de preuve d'optimalité, nous avons comparé ses performances avec la méthode de résolution présentée dans le chapitre 2 sur les instances cb5.250 de la `OR-Library`. Rappelons que la méthode de résolution du chapitre 2 consiste à calculer un minorant de départ avec l'heuristique de recherche locale RL^{tabou} [67] puis à résoudre l'instance par une énumération implicite. Nous avons découpé nos expérimentations en deux parties :

1. *Calcul de minorants :* nous avons comparé Resolution search avec RL^{tabou} pour le calcul de minorants. La colonne *BI* donne le minorant obtenu par RL^{tabou}, la colonne RL^{tabou} donne le temps de calcul en secondes requis par cette méthode pour trouver BI et la colonne *rs* donne le temps de calcul en secondes requis par Resolution search pour obtenir un minorant au moins aussi bon que BI.
2. *Temps de preuve :* nous avons exécuté Resolution search et l'énumération implicite en considérant le minorant donné par RL^{tabou} comme minorant de départ. La colonne *opt.* donne la valeur optimale de l'instance, la colonne *enum* donne le temps de calcul en secondes requis par l'énumération implicite pour résoudre l'instance à partir de BI et la colonne *rs* donne le temps de calcul en secondes requis par Resolution search pour résoudre l'instance à partir de BI.

En ce qui concerne le calcul de minorants, l'implémentation de Resolution search proposée est nettement plus rapide que l'heuristique de recherche locale RL^{tabou}. Seulement 5 minorants sur les 30 ont été trouvés en un temps plus rapide par RL^{tabou}. Si l'on compare la somme des temps pour toutes les instances, Resolution search trouve des minorants au moins aussi bons que ceux trouvés par RL^{tabou} en un temps 40% inférieur. En terme de preuve d'optimalité, Resolution search se comporte moins bien que l'énumération implicite. En effet, si l'on compare la somme des temps de preuve pour toutes les instances, l'énumération implicite met environ 88% de temps en moins que Resolution search pour prouver l'optimalité des instances traitées.

3.10 Conclusion

Dans ce chapitre, nous avons présenté et détaillé le mécanisme de Resolution search. Nous avons vu, par des expérimentations, que la phase de remontée telle que décrite dans le papier original dégradait de manière significative les performances de l'algorithme dans le cas d'une implémentation simple pour le MKP. Cependant, les expérimentations nous ont montré que si l'on ne prend pas en compte la phase de remontée, Resolution search se comporte de manière similaire à l'énumération implicite et qu'elle est tout autant influencée par les stratégies de branchement. Nous avons vu que l'efficacité de la méthode peut être améliorée si l'on prend en compte un critère de remise en cause plus pertinent ou encore si l'on cherche à générer des obstacles plus petits par l'intermédiaire des coûts

réduits. Une dernière expérimentation nous a montré que Resolution search pouvait être très performante pour le calcul de minorants mais qu'elle résolvait moins rapidement que l'énumération implicite l'ensemble des instances cb5.250 si un bon minorant de départ est connu. Nous montrons dans le chapitre suivant qu'il est possible d'améliorer la recherche d'obstacles minimaux, mais également qu'il est intéressant d'hybrider les deux méthodes afin d'exploiter l'habilité de Resolution search à calculer de bons minorants rapidement et la capacité d'élagage de l'arbre de recherche de l'énumération implicite.

TAB. 3.6 – Résultats obtenus par l'énumération implicite sur les instances cb5.500

instance	Calcul de minorant			Preuve d'optimalité		
	BI	rs	RL^{tabou}	opt	rs	enum
cb5.250_0	59312	**44**	80	59312	5	**0**
cb5.250_1	61472	**42**	79	61472	45	**3**
cb5.250_2	62130	**9**	61	62130	6	**1**
cb5.250_3	59453	**3**	106	59463	389	**93**
cb5.250_4	58951	**38**	95	58951	101	**6**
cb5.250_5	60062	**0**	78	60077	82	**28**
cb5.250_6	60396	**3**	98	60414	91	**6**
cb5.250_7	61449	**1**	101	61472	175	**36**
cb5.250_8	61885	**0**	73	61885	60	**3**
cb5.250_9	58959	**2**	87	58959	5	**0**
cb5.250_10	109086	**2**	96	109109	72	**15**
cb5.250_11	109841	117	**97**	109841	28	**2**
cb5.250_12	108508	**34**	72	108508	55	**3**
cb5.250_13	109378	**66**	98	109383	137	**12**
cb5.250_14	110718	**43**	95	110720	356	**21**
cb5.250_15	110256	**0**	91	110256	100	**5**
cb5.250_16	109040	301	**79**	109040	63	**3**
cb5.250_17	109042	308	**95**	109042	238	**14**
cb5.250_18	109971	**71**	100	109971	89	**5**
cb5.250_19	107058	263	**90**	107058	107	**6**
cb5.250_20	149659	**0**	91	149665	87	**5**
cb5.250_21	155940	**8**	92	155944	60	**3**
cb5.250_22	149334	**3**	95	149334	97	**24**
cb5.250_23	152130	**49**	103	152130	33	**2**
cb5.250_24	150353	**25**	98	150353	70	**4**
cb5.250_25	150045	105	**73**	150045	3	**0**
cb5.250_26	148607	**3**	75	148607	2	**0**
cb5.250_27	149772	**14**	95	149782	133	**8**
cb5.250_28	155061	**0**	91	155075	11	**1**
cb5.250_29	154662	**35**	95	154668	56	**4**

Chapitre 4

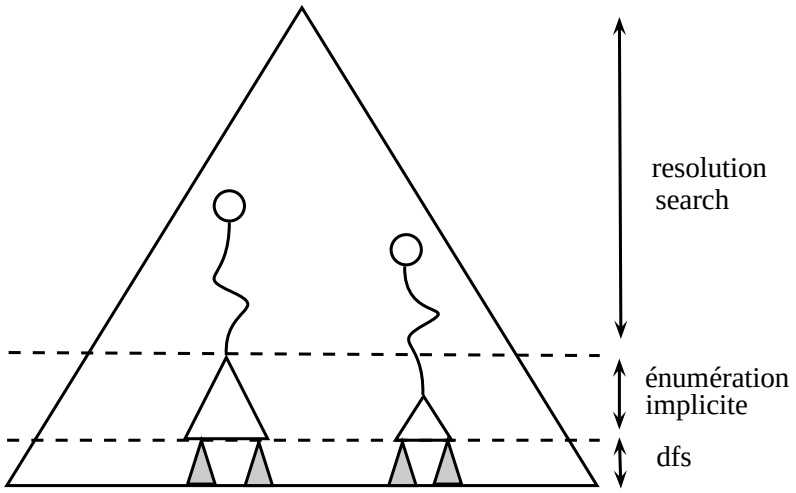
Coopération entre Resolution search et l'énumération implicite pour le sac à dos multidimensionnel en 0–1

Dans ce chapitre, nous présentons un méthode de résolution du MKP combinant Resolution search et une variante de l'algorithme d'énumération implicite présenté dans le chapitre 2. Cette méthode s'appuie également sur la prise en compte des coûts réduits à l'optimum de la relaxation continue et sur la décomposition de l'espace en hyperplans. La coopération est axée sur une exploration en trois niveaux de l'arbre de recherche : les branches hautes sont énumérées par Resolution search, les branches médianes par l'énumération implicite et les branches basses par l'algorithme de backtracking. Nous montrons que la structure de Resolution search permet d'explorer partiellement et de manière répétée les différents sous-problèmes associés aux hyperplans, ceci afin de diversifier la recherche. Nous proposons également une méthode alternative à la phase de remontée qui utilise des relations d'implication entre noeuds de l'arbre de recherche pour identifier des obstacles minimaux. Nous montrons que cette phase de remontée, baptisée *phase de remontée implicite*, accélère le processus de recherche dans le cas de notre implémentation fondée sur les coûts réduits. Les expériences numériques prouvent l'efficacité de cet algorithme : il a des performances comparables aux meilleures heuristiques connues en terme de calcul de minorant et résout des instances de taille moyenne en un temps plus rapide que les meilleures méthodes exactes connues sur ces mêmes instances. De plus, il résout les instances de grande taille à 10 contraintes et 500 variables de la `OR-Library`. Les valeurs optimales de ces instances étaient inconnues jusqu'à présent. Le travail présenté dans ce chapitre a fait l'objet d'une soumission à une revue internationale [9].

4.1 Principe général

D'un point de vue global, les branchements étant réalisés suivant l'ordre décroissant des coûts réduits, l'arbre de recherche est exploré suivant trois niveaux : Resolution search explore les branches hautes (variables à fort coût réduit), l'énumération implicite explore les branches médianes (variables à coût réduit moyen) et l'algorithme de backtracking explore les banches basses (variables de plus faible coût réduit et variables de base). Les deux méthodes coopèrent de la manière suivante : Resolution search fournit des sous–problèmes à l'énumération implicite qui, en les résolvant, permet d'obtenir des obstacles de petite taille utilisés par Resolution search pour guider l'exploration vers des zones prometteuses de l'espace de recherche.

Fig. 4.1 – Exploration de l'arbre de recherche en trois niveaux



4.2 Contributions apportées à Resolution search

Dans cette section, nous présentons deux modifications apportées à Resolution search. La première est une version à itérations limitées de l'algorithme qui permet d'explorer itérativement chaque sous–problème lié à la décomposition par hyperplans et la deuxième est une modification de la phase de remontée fondée sur des relations d'implication entre nœuds de l'arbre de recherche.

4.2.1 Exploration itérative de l'espace de recherche

Comme nous l'avons mentionné dans le chapitre 2, l'ordre dans lequel les hyperplans sont explorés influence fortement l'efficacité de la mise à jour de la borne inférieure. Nous allons voir que la structure de Resolution search présente un avantage intéressant qui permet de palier à ce problème.

Dans le chapitre 3, nous avons montré que la famille path–like $\mathscr{F}$ constitue un résumé des explorations passées : après chaque exécution de la boucle principale, $\mathscr{F}$ nous donne l'état de la recherche et le nœud suivant de l'exploration $u(\mathscr{F})$. Cette particularité permet de pouvoir interrompre le processus de recherche à un moment donné et de le récupérer ultérieurement sans perte d'information. De cette façon, il est possible de résoudre le problème P en effectuant de manière répétée un nombre d'itérations limité pour chaque sous–problème $P(k)$ jusqu'à ce que tous les sous–problèmes soient résolus. L'ordre d'exploration n'a alors plus d'importance car il n'est pas nécessaire qu'un sous–problème soit résolu pour commencer la résolution du sous–problème suivant. Les algorithmes 4.1 et 4.2 illustrent cette façon itérative de résoudre le problème.

Algorithme 4.1 – Algortithme Resolution search itératif

```
IRS(P,Nb_Iter, BI,F)
{
   iter = 0 ;
   while(iter < Nb_Iter) {
      try = obstacle(P, u(F),BI,S) ;
      if(try > BI)  BI = try ;
      ajouter S à F et mettre à jour F ;
      if((*,*,...,*) ∈ F)   Break ;
      iter++;
   }
}
```

Algorithme 4.2 – Exploration des hyperplans

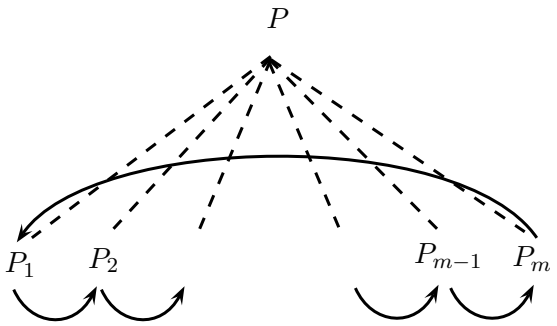
```
resoudre_MKP() {
   BI = glouton() ;
   Calculer les bornes k_min et k_max ;
   Définir K = {k_min , ... , k_max} ;
   for(k = k_min, ..., k_max) CFamily[k] = ∅
   While (K ≠ ∅) {
      Choisir k ∈ K ;
      Choisir Nb_Iter_k ≥ 1 ;
      IRS(P(k),Nb_iter_k,BI,CFamily[k]) ;
      if((*,*,...,*) ∈ CFamily[k])   K = K − {k} ;
   }
}
```

La fonction `IRS` de l'algorithme 4.1 prend en arguments : un problème P, un nombre d'itérations limité `Nb_Iter`, un minorant `BI` et une famille path–like $\mathscr{F}$. Cet algorithme consiste à effectuer un nombre `Nb_Iter` d'itérations de Resolution search.

La fonction `resoudre_MKP()` de l'algorithme 4.2 consiste à résoudre le problème MKP en utilisant cette version iterative de Resolution search. L'algorithme calcule une borne inférieure gloutonne afin de déduire les bornes k_{min} et k_{max}, puis effectue un certain nombre d'itérations de `IRS` sur chaque sous–problème $P(k)$. La variable CFamily[k] correspond à la famille path–like de `IRS` pour le problème $P(k)$. Lorsque l'un des problèmes $P(k)$ est résolu, c'est–à–dire que $(*, *, ..., *) \in$ CFamily[k], il est éliminé de la recherche. Le problème P est alors résolu lorsque tous les sous–problèmes $P(k)$ sont éliminés.

Ce type d'exploration peut être étendu à une quelconque décomposition $P_1, P_2, ..., P_m$ du problème P original. Il peut par exemple être utilisé pour la résolution itérative de sous–problèmes pour lesquels certaines variables sont fixées. La figure 4.2 illustre cette approche de résolution.

Fig. 4.2 – Exploration itérative des sous–problèmes



4.2.2 Phase de remontée implicite

Dans le papier original de Resolution search, Chvátal propose de libérer une à une les variables précédemment instanciées dans la phase de descente (sauf la dernière) et de vérifier à chaque fois la faisabilité de la solution partielle correspondante en exécutant la fonction `oracle` (cf. chapitre 3). Cette phase de remontée correspond à la portion de code suivante de l'algorithme 3.4 :

```
while(la pile n'est pas vide){
   dépiler j ;
   S_j = * ;
   if(oracle(S) > BI) S_j = u_j^+ ;
}
```

Nous avons montré par des expérimentations que cette portion de l'algorithme ralentit considérablement le processus de recherche (cf. chapitre 3 section 3.9.1). Nous proposons un autre moyen d'identifier la clause S que nous appelons *phase de remontée implicite*.

Contrairement à la phase de remontée originale, cette méthode ne cherche pas à libérer des variables une fois l'échec rencontré, mais identifie, lors de la phase de descente, un ensemble d'instanciations qui ne seront pas contenues dans le futur obstacle. Cette méthode est basée sur des relations d'implication existantes entre certains nœuds de l'arbre de recherche.

Supposons que l'instanciation des variables dans le nœud u implique la fixation de variables supplémentaires constituant le nœud v. En d'autres termes, supposons que $u \Rightarrow v$. Si, pendant la phase de descente, u est étendu en $u^+ = u \cup v \cup w$ et que u^+ viole une ou plusieurs contraintes du problème, la clause S des affectations responsables de l'échec est réduite à $u \cup w$.

Proposition 4.1 *Soit u, v et w des vecteurs de $\{0,1,*\}^n$. Si $u \Rightarrow v$ et si $u \cup v \cup w$ est un obstacle, alors $u \cup w$ est un obstacle.*

Preuve Nous allons prouver la proposition par induction sur la longueur de v et par l'absurde. Supposons que v soit de longueur 1 tel que $v = \{x_j\}$, donc $u' = u \cup \{\bar{x}_j\}$ est un obstacle puisque $u \Rightarrow v$ et $u'' = u \cup \{x_j\} \cup w$ est un obstacle par hypothèse. Alors, la résolvante de u' et u'' qui est $u' \nabla u'' = u \cup w$ est un obstacle. Supposons à présent que la proposition soit vraie pour un vecteur v de longueur k, nous allons prouver que la proposition est toujours vraie si v est de longueur $k+1$. Soit $v = v' \cup \{x_j\}$, alors $u' = u \cup v' \cup \{\bar{x}_j\}$ est un obstacle et $u'' = u \cup v' \cup \{\bar{x}_j\} \cup w$ est également un obstacle. Donc la résolvante de u' et u'' qui est $u' \nabla u'' = u \cup v' \cup w$ est un obstacle et par hypothèse d'induction, $u \cup w$ est un obstacle. □

Bien que la phase de remontée implicite puisse remplacer la phase de remontée originale de Resolution search, son efficacité dépend du problème résolu et des relations d'implication qui peuvent être déduites entre les nœuds de l'arbre de recherche.

Dans le cas de la résolution du MKP, nous proposons une implémentation fondée sur les relations d'implication qui peuvent être déduites de la contrainte des coûts réduits. Dans le cas où le nœud de départ $u(\mathscr{F})$ donné à la fonction `obstacle` contient des instanciations des variables à l'opposé de leur valeur optimale, *gap* est diminuée de la valeur des coûts réduits correspondants et cette diminution implique la fixation des variables non instanciées dans $u(\mathscr{F})$ ayant un coût réduit supérieur à *gap*. D'après la proposition 4.1, les variables ainsi fixées à cause de la diminution de *gap* engendrée par $u(\mathscr{F})$ ne sont alors pas prises en compte dans la clause S. L'algorithme 4.3 détaille l'application de la phase de remontée implicite dans la fonction `obstacle`.

Algorithme 4.3 – Fonction obstacle avec phase de remontée implicite pour le MKP

```
obstacle(P,u,BI,S)
{
   gap = v(P̄)− BI ;
   for(j = 1,2,...,n)
      if(u_j = 1 − x̄_j) gap = gap − c̄_j ;
   u+ = u ;S = u+ ;
   borne = oracle(u+) ;
   if(borne > BI){
      //Phase de remontée implicite
      for(j = 1,2,...,n)
         if(u_j = ∗ et c̄_j > gap) u+_j = x̄_j ;
      //Phase de descente
      while(u+ ∉ {0,1}^n) {
         choisir un indice j tel que u+_j = ∗ et une valeur v dans {0,1} ;
         u+_j = v ;S_j = v ;
         borne = oracle(u+) ;
         if(borne ≤ BI) break ;
      }
      if(borne > BI) BI = borne ;
   }
   return borne ;
}
```

Nous avons mené une expérimentation de Resolution search avec et sans phase de remontée implicite afin d'en évaluer l'efficacité. Le tableau 4.1 donne les résultats obtenus sur les instances cb5.100 de la `OR-Library`. La colonne *opt.* donne le nombre moyen d'appels à `oracle` nécessaires à l'obtention de la valeur optimale et la colonne *preuve* donne le nombre moyen d'appels à `oracle` nécessaires à la preuve d'optimalité.

Tab. 4.1 – Impact de la phase de remontée implicite

Instance	sans remontée implicite		avec remontée implicite	
	opt.	preuve	opt.	preuve
0-9	69286	145395	**55726**	**111865**
10-19	71286	155152	**57736**	**122944**
20-29	17414	45696	**13891**	**39376**

On constate une nette diminution du nombre moyen d'appels à `oracle` pour l'obtention de la valeur optimale mais également pour la preuve d'optimalité. Si l'on compare la somme des appels à `oracle` pour l'ensemble des 30 instances cb5.100, on constate une diminution de l'ordre de $19,3\%$ pour l'obtention de la valeur optimale et une diminution de l'ordre de $20,8\%$ pour l'obtention de la preuve d'optimalité.

4.3 Description du processus d'exploration

Dans cette section, nous décrivons le principe d'exploration de la fonction `obstacle`. Nous détaillons l'application de la phase de remontée implicite ainsi que l'hybridation avec l'algorithme d'énumération implicite.

Partant du nœud $u(\mathscr{F})$ fourni par la famille path–like $\mathscr{F}$, `obstacle` remplace pas–à–pas les composantes $*$ de $u(\mathscr{F})$ par 0 ou 1 de manière à construire le nœud u^+. Si u^+ devient un obstacle (alors noté u^*), la procédure cherche une clause minimale S telle que $S \sqsubseteq u^*$ et S est un obstacle. Afin d'illustrer le déroulement de la fonction `obstacle`, nous considérons le problème P^d suivant :

$$
\begin{array}{lrcrcrcrcrcl}
(P^d)\ \text{Maximiser} & 10x_1 & + & 10x_2 & + & 8x_3 & + & 8x_4 & + & 7x_5 & & \\
\text{sujet à} & 5x_1 & + & 2x_2 & + & 10x_3 & + & 1x_4 & + & 3x_5 & \leq & 10 \\
& 2x_1 & + & 11x_2 & + & 2x_3 & + & 10x_4 & + & x_5 & \leq & 11 \\
& x = {}^T(x_1, x_2, x_3, x_4, x_5) \in \{0,1\}^5 & & & & & & & & & &
\end{array}
$$

Supposons que l'on cherche à résoudre le problème $P^d(2)$ qui correspond au problème P^d avec la contrainte additionnelle $(1 \cdot x = 2)$. La résolution de la relaxation linéaire de $P^d(2)$ nous donne une solution optimale $\bar{x}$ de valeur $\texttt{BS} = 19.55$ et un vecteur de coûts réduits $\bar{c}$ tels que :

$$\bar{x} = (1, 0.78, 0.22, 0, 0) \qquad \text{et} \qquad \bar{c} = (2, 0, 0, -1.78, -0.78)$$

Supposons que nous connaissions un minorant $\texttt{BI} = 16$ du problème, alors la contrainte des coûts réduits s'écrit

$$2(1 - x_1) + 1.78x_4 + 0.78x_5 \leq 2$$

Au début de la procédure, la relaxation linéaire du problème est résolue pour chaque hyperplan disponible dans le but de fournir l'information nécessaire à la génération de la contrainte des coûts réduits.

La première étape, appelée *Phase de vérification*, consiste à vérifier la faisabilité de $u(\mathscr{F})$. Si une contrainte est violée, la vérification s'arrête et l'affectation partielle correspondante, représentant un obstacle, est mémorisée dans $\mathscr{F}$ sous forme de clause. Simultanément, la contrainte des coûts réduits est vérifiée et la valeur *gap* est mise à jour : pour chaque variable hors base fixée à l'opposé de sa valeur optimale $(1 - \bar{x}_j)$, son coût réduit $(\bar{c}_j)$ est soustrait à *gap* et si $gap < 0$, l'affectation partielle courante est un obstacle. Dans ce cas, S est uniquement composée des instanciations des variables fixées à l'opposé de leur valeur optimale dans $u(\mathscr{F})$. Par exemple, supposons que lors de la résolution de $P^d(2)$, $u(\mathscr{F}) = (0, 0, 0, 1, *)$, nous avons gap $= -1.78$ donc $u(\mathscr{F})$ est un obstacle. La solution partielle $(0, *, *, 1, *) \sqsubseteq (0, 0, 0, 1, *)$ qui viole la contrainte des coûts réduits est également un obstacle et dans ce cas $S = x_1\bar{x}_4$ est ajoutée à $\mathscr{F}$.

Nous passons ensuite à la deuxième étape qui constitue l'application de la phase de remontée implicite décrite en section 4.2.2. Cette phase consiste à fixer toutes les variables libres hors base de coût réduit strictement supérieur à *gap* à leur valeur optimale. Ces variables doivent en effet être fixées à leur valeur optimale pour satisfaire la contrainte des coûts réduits. Fixer ces variables à leur valeur optimale n'est donc qu'une conséquence des instanciations des variables dans $u(\mathscr{F})$. Selon la proposition 4.1, elles peuvent ne pas être incluses dans S. Pour illustrer ce processus, supposons que `obstacle` commence avec $u(\mathscr{F}) = (0, *, *, *, *)$, x_1 étant fixé à $(1 - \bar{x}_1)$, on a $gap = 0$. Puisque $\bar{c}_4 = 1.78$ est supérieur à *gap*, x_4 doit être fixée à $\bar{x}_4 = 0$. Nous avons donc $u(\mathscr{F}) = (0, *, *, *, *)$ implique $u^+ = (0, *, *, 0, *)$ et $x_4 = 0$ ne sera pas inclue dans S.

La phase suivante constitue la phase de descente (ou *waxing phase*) telle que décrite dans le papier original [14]. Cette phase consiste à assigner des valeurs aux variables libres jusqu'à atteindre un obstacle. La stratégie de branchement choisie consiste à sélectionner la variable libre de plus grande valeur absolue de coût réduit et à lui assigner sa valeur optimale $\bar{x}_j$. Chaque fois qu'un branchement est réalisé, la réalisabilité de l'affectation partielle courante est vérifiée et en cas d'echec, la phase de descente s'arrête et la clause S correspondante est ajoutée à $\mathscr{F}$. Si nous considérons l'exemple précédent, $u^+ = (0, *, *, 0, *)$ après la phase de remontée implicite. Puisque x_5 est la variable libre de plus grande valeur absolue de coût réduit ($|\bar{c}_5| = 0,78$), nous la fixons à sa valeur optimale ($u_5^+ = 0$).

Lorsque le nombre de variables libres restant est inférieur ou égal à un paramètre Δ donné, la phase de descente s'arrête et le sous–problème correspondant est résolu par l'énumération implicite. Ce sous–problème inclut les variables libres de faible coût réduit et les variables de base. L'énumération implicite explorant entièrement le sous–arbre correspondant aux variables libres, la clause S générée ne contient aucun des branchements réalisés pendant cette phase. Seules les branchement réalisés pendant la phase de vérification et / ou la phase de descente sont considérées pour la construction de la clause S. Dans l'exemple ci–dessus, l'énumération des variables restantes x_2 et x_3 donne les configurations suivantes :

$u^* = (0, 0, 0, 0, 0)$, $c \cdot u^* \leq \texttt{BI}$,

$u^* = (0, 0, 1, 0, 0)$, $c \cdot u^* \leq \texttt{BI}$,

$u^* = (0, 1, 0, 0, 0)$, $c \cdot u^* \leq \texttt{BI}$,

$u^* = (0, 1, 1, 0, 0)$, $A \cdot u^* > b$: irréalisable.

Puisqu'aucune de ces configurations n'améliore la meilleure solution connue, la clause $S = \bar{x}_1 \bar{x}_5$ est ajoutée à $\mathscr{F}$. Notons que Δ peut être mise à jour dynamiquement durant la recherche, selon que l'on veut favoriser la partie Resolution search ou la partie énumération implicite.

L'algorithme utilisé pour énumérer les variables du sous–problème constitué des variables de faible coût réduit et des variables de base est similaire à l'énumération implicite présentée dans le chapitre 2.

L'unique modification apportée a été de retirer à cet algorithme la propagation des coûts réduits effectuée à chaque nœud (cf. chapitre 2, section 2.4.3). En effet, dans le cas de la résolution de ces sous–problèmes constitués essentiellement de variables à faible coût réduit, l'expérimentation nous a montré que cette propagation perd de son efficacité et a tendance à ralentir le processus de recherche. Comme mentionné précédemment, cet algorithme oriente la recherche vers les solutions les plus à même de violer la contrainte des coûts réduits. La politique de branchements à chaque nœud u peut se résumer de la manière suivante :

1. Résoudre le problème $\bar{P}(k, u)$.
2. Brancher sur la variable libre hors base (x_j) de plus grande valeur absolue de coût réduit ($|\bar{c}_j|$) en la fixant à l'opposé de sa valeur optimale ($1 - \bar{x}_j$).
3. Mettre à jour $gap = \lfloor \texttt{BS}(u) \rfloor - \texttt{BI} - 1$ par $gap = gap - \bar{c}_j$.
4. Fixer toutes les variables hors base de coût réduit strictement supérieur à gap à leur valeur optimale.

A chaque nœud, la règle de branchement cherche à provoquer la violation de la contrainte des coûts réduits de manière à élaguer au plus tôt l'arbre de recherche. Nous avons montré dans le chapitre 2 que cette méthode est efficace uniquement lorsqu'un bon minorant du problème est connu. Dans le cas de l'hybridation avec Resolution search, elle correspond bien à la stratégie générale : utiliser Resolution search pour explorer l'espace de recherche de manière pertinente et l'énumération implicite pour résoudre efficacement des sous–problèmes de petite taille.

4.4 Expériences numériques

Les expériences numériques ont été réalisées sur les instances du jeu Chu et Beasley de la `OR-Library`. Les résultats sont comparés avec ceux obtenus par l'énumération implicite du chapitre 2 ainsi qu'avec le solveur commercial CPLEX 9.2. Le paramètre Δ est incrémenté toutes les 100 itérations partant de 20 jusqu'à 60, et réinitialisé à 20 si la borne inférieure est améliorée. Les résultats obtenus sur les instances cb10.250, cb5.500 et cb10.500 sont présentés dans les tableaux 4.2, 4.3 et 4.4. La description des données par colonne est la suivante :

- *Instance* est le nom de l'instance.
- z^{opt} est la valeur optimale.
- *opt.* est le temps de calcul requis en secondes pour obtenir une solution optimale.
- *preuve* est le temps de calcul requis en secondes pour prouver l'optimalité de la meilleure solution trouvée.

La colonne *RSBB* correspond aux résultats obtenus avec l'hybridation de Resolution search et de l'énumération implicite, la colonne *enum [69]* correspond aux résultats obtenus avec l'énumération implicite présentée dans le chapitre 2 et la colonne *CPLEX* correspond

aux résultats obtenus par CPLEX 9.2 (ERROR si CPLEX a dépassé la capacité de mémoire). Les résultats obtenus sur les instances à 10 contraintes et 250 variables sont exposés dans le tableau 4.2 et ceux obtenus sur les instances à 5 contraintes et 500 variables sont exposés dans le tableau 4.3. Dans le tableau 4.4, correspondant aux instances à 10 contraintes et 500 variables, la colonne $z^{opt} - \underline{z}$ donne la différence entre la valeur optimale trouvée par notre algorithme et le meilleur minorant connu jusqu'à présent : (vv) indique qu'il y a eu une amélioration par rapport aux résultats de Vasquez et Vimont [68] et (wh) indique qu'il y a eu une amélioration par rapport aux résultats de Wilbaut et Hanafi [72]. Cette méthode résout l'ensemble des instances cb10.250 et cb5.500 plus rapidement que l'énumération implicite présentée dans le chapitre 2 et que CPLEX 9.2. De plus, elle résout l'ensemble des instances cb10.500 dont les valeurs optimales étaient inconnues jusqu'à présent.

Tab. 4.2 – Résultats obtenus par l'algorithme hybride sur les instances cb10.250

Instance	z^{opt}	RSBB		enum [69]	CPLEX
		opt.	preuve	preuve	preuve
cb10.250_0	59187	1	**3326**	5126	ERROR
cb10.250_1	58781	325	**1510**	43910	4369
cb10.250_2	58097	2	**795**	1509	6746
cb10.250_3	61000	1906	**6430**	9080	ERROR
cb10.250_4	58092	738	**20749**	28789	ERROR
cb10.250_5	58824	6	**747**	1099	7394
cb10.250_6	58704	0	**647**	1052	8255
cb10.250_7	58936	3426	**58294**	87329	ERROR
cb10.250_8	59387	335	**2391**	4428	ERROR
cb10.250_9	59208	0	**4446**	6902	ERROR
cb10.250_10	110913	560	**3593**	4952	ERROR
cb10.250_11	108717	630	**3521**	7465	ERROR
cb10.250_12	108932	4	**2141**	3382	ERROR
cb10.250_13	110086	91	**9573**	15091	ERROR
cb10.250_14	108485	0	**1585**	2306	9869
cb10.250_15	110845	94	**4130**	5954	ERROR
cb10.250_16	106077	34	**5130**	7867	ERROR
cb10.250_17	106686	816	**3362**	5538	ERROR
cb10.250_18	109829	17	**2930**	4817	ERROR
cb10.250_19	106723	24	**807**	1384	5716
cb10.250_20	151809	353	**584**	973	4695
cb10.250_21	148772	0	**1559**	2284	ERROR
cb10.250_22	151909	0	**292**	828	5048
cb10.250_23	151324	318	**563**	5489	2234
cb10.250_24	151966	533	**812**	932	5727
cb10.250_25	152109	3	**323**	818	2826
cb10.250_26	153131	7	**50**	239	374
cb10.250_27	153578	0	**3279**	8469	ERROR
cb10.250_28	149160	109	**363**	1276	1638
cb10.250_29	149704	0	**413**	826	2487

TAB. 4.3 – Résultats obtenus par l'algorithme hybride sur les instances cb5.500

Instance	z^{opt}	RSBB		enum [69]	CPLEX
		opt.	preuve	preuve	preuve
cb5.500_0	120148	60	**109**	1331	8001
cb5.500_1	117879	9	**15**	1034	855
cb5.500_2	121131	5	**35**	1112	4687
cb5.500_3	120804	14	**47**	630	6050
cb5.500_4	122319	5	**27**	558	1578
cb5.500_5	122024	45	**67**	932	3678
cb5.500_6	119127	4	**51**	851	6937
cb5.500_7	120568	7	**27**	804	2672
cb5.500_8	121586	48	**77**	1172	ERROR
cb5.500_9	120717	1	**44**	2783	5756
cb5.500_10	218428	53	**63**	838	1457
cb5.500_11	221202	1	**11**	2146	905
cb5.500_12	217542	221	**232**	1634	3728
cb5.500_13	223560	1	**82**	1211	5009
cb5.500_14	218966	8	**11**	486	485
cb5.500_15	220530	21	**40**	814	3122
cb5.500_16	219989	10	**22**	593	1211
cb5.500_17	218215	1	**18**	811	1096
cb5.500_18	216976	1	**37**	697	2373
cb5.500_19	219719	6	**67**	969	2522
cb5.500_20	295828	1	**5**	767	47
cb5.500_21	308086	20	**44**	674	475
cb5.500_22	299796	1	**10**	783	187
cb5.500_23	306480	8	**22**	620	1182
cb5.500_24	300342	1	**27**	610	204
cb5.500_25	302571	10	**16**	634	988
cb5.500_26	301339	5	**6**	466	366
cb5.500_27	306454	3	**6**	558	148
cb5.500_28	302828	3	**18**	724	367
cb5.500_29	299910	59	**93**	560	478

Les résultats obtenus illustrent les points positifs suivants de notre approche : (i) en dépit des temps de calculs parfois très élevés, notre algorithme n'a jamais dépassé la capacité de mémoire et (ii) les temps de calcul requis pour trouver les valeurs optimales son relativement faibles. En effet, nous avons comparé les minorant obtenus par notre approche avec ceux obtenus par l'heuristique de Wilbaut et Hanafi [72] en un temps limité à 2 heures[1]. On considère qu'une méthode est plus efficace qu'une autre si un même minorant a été trouvé en un temps plus rapide ou si un meilleur minorant a été trouvé dans le temps imparti.

[1]D'autres heuristiques performantes ont également été proposées par Vasquez et Hao [67] et Vasquez et Vimont [68] cependant les temps d'obtention des meilleurs minorants sont quelquefois long (entre 300 secondes et 25 heures) alors que les résultats publiés par Wilbaut et Hanafi sont ceux obtenus en 2 heures de temps de calcul. C'est pourquoi nous comparons nos résultats avec cette approche

Tab. 4.4 – Résultats obtenus par l'algorithme hybride sur les instances cb10.500

Instance	z^{opt}	opt.	preuve	$z^{opt} - \underline{z}$
cb10.500_0	117821	5525	2041200	+**10**(vv)
cb10.500_1	119249	246521	982776	+**17**(vv)
cb10.500_2	119215	4200	2764800	0
cb10.500_3	118829	170695	322744	+**4**l(wh)
cb10.500_4	116530	19394	9108000	+**16**(wh)
cb10.500_5	119504	8375	676837	0
cb10.500_6	119827	9892	461009	0
cb10.500_7	118344	582278	646591	+**11**(wh)
cb10.500_8	117815	310868	791701	0
cb10.500_9	119251	11423	1277833	0
cb10.500_10	217377	18	1856970	0
cb10.500_11	219077	2083	1575519	0
cb10.500_12	217847	18	20129	0
cb10.500_13	216868	67	376181	0
cb10.500_14	213873	7345	4975200	+**14**(vv)
cb10.500_15	215086	123	158186	0
cb10.500_16	217940	48419	130088	0
cb10.500_17	219990	543060	1255820	0
cb10.500_18	214382	45998	208335	+**7**(vv)
cb10.500_19	220899	743	76686	0
cb10.500_20	304387	23938	29756	0
cb10.500_21	302379	58	30392	0
cb10.500_22	302417	241986	380031	+**1**(vv)
cb10.500_23	300784	3345	13682	0
cb10.500_24	304374	683	60619	0
cb10.500_25	301836	107128	111286	0
cb10.500_26	304952	169	66919	0
cb10.500_27	296478	4122	33730	0
cb10.500_28	301359	29488	140984	0
cb10.500_29	307089	4324	16085	0

Notre méthode est plus efficace que l'heuristique de Wilbaut et Hanafi pour l'ensemble des instances cb5.500 et elle est plus efficace pour 16 des 30 instances cb10.500.

4.5 Amélioration du calcul de minorants

Nous avons expérimenté une méthode consistant à renforcer la contrainte des coûts réduits sur chaque hyperplan afin d'améliorer le calcul de minorants. Pour ce faire, nous proposons de réduire le membre de droite de la contrainte des coûts réduits $gap = \lfloor \mathtt{BS} \rfloor - \mathtt{BI} - 1$ d'une valeur ρ puis de diminuer ρ successivement après un certain nombre d'itérations jusqu'à obtenir le membre de droite initial. L'intérêt de cette démarche est que la diminution du membre de droite de la contrainte des coûts réduits favorise la fixation de variables et l'élagage de l'arbre de recherche.

Notons qu'à chaque fois que la valeur ρ est modifiée, les familles path–like de chaque hyperplan sont vidées de leur clauses. Cependant, afin d'éviter la redondance de la recherche, certaines clauses sont conservées : pour les 3 hyperplans de plus grande valeur $v(\bar{P}(k))$, nous conservons les clauses C_i pour lesquelles $\texttt{oracle}(C_i) \leq \texttt{BI}$. Expérimentalement, nous avons testé cette méthode sur les instances cb10.500 de la `OR-Library`. Nous avons conclu des premières expérimentations qu'initialiser l'écart $\rho = 500$ puis le diminuer de 50 toutes les 200 secondes donnait des résultats intéressants. Il est probable qu'un autre paramétrage donne de meilleurs résultats mais cette expérimentation a simplement pour but de donner une première évaluation de cette méthode.

Le tableau 4.5 montre les résultats obtenus en comparaison avec l'heuristique de Wilbaut et Hanafi [72]. Dans ce tableau 4.5, la colonne *BI* correspond au meilleur minorant trouvé et *t(s)* est le temps de calcul requis pour trouver BI. Les colonnes *Wilbaut, Hanafi [72]* correspondent aux résultats trouvés par l'heuristique de Wilbaut et Hanafi, *RSBB* correspond aux résultats obtenus par la méthode hybride présentée dans ce chapitre en limitant son temps d'exécution à 2 heures et *RSBBC* correspond aux résultats obtenus en faisant varier le membre de droite de la contrainte des coûts réduits. Les valeurs en gras indiquent une amélioration soit en temps (un même minorant a été trouvé en un temps plus rapide) soit en valeur (un meilleur minorant a été trouvé dans le temps imparti).

Ces résultats montrent les possibilités d'amélioration de cette méthode hybride pour la calcul de minorants. Avec cette première méthode, nous avons en effet obtenu une amélioration de 25 instances sur les 30 instances cb10.500 par rapport à l'heuristique de Wilbaut et Hanafi.

4.6 Conclusion

Nous avons proposé, dans ce chapitre, certaines contributions à Resolution search. La première contribution concerne une modification de la phase de remontée originale, fondée sur des relations d'implication qui peuvent être déduites entre certains nœuds de l'arbre de recherche. Cette phase de remontée implicite, permet de procéder à l'identification d'obstacles minimaux à moindre coût par rapport à la phase de remontée originale. Nous avons vu expérimentalement qu'elle offre une réduction sensible du nombre d'appels à `oracle` lors de la résolution d'instances de MKP. Nous avons également montré que la structure de Resolution search permet d'améliorer la diversification de la recherche en explorant partiellement et de manière répétée plusieurs sous–problèmes du problème original. Nous avons proposé une méthode hybride qui combine Resolution search à un algorithme d'énumération implicite inspiré des travaux du chapitre 2. Cette hybridation constitue une réelle coopération entre Resolution search, qui guide la recherche vers des zones prometteuses de l'espace, et l'énumération implicite qui, en résolvant efficacement des sous–problèmes de petite taille, fournit des obstacles intéressants à Resolution search. Les expérimentations ont montré que la méthode résultante est capable de résoudre les instances cb10.250 et cb5.500

en un temps plus rapide que les meilleures méthodes exactes connues sur ces instances et d'obtenir les preuves d'optimalité des instances cb10.500 qui étaient inconnues jusqu'à présent. Nous avons vu qu'il était possible, en resserrant la contrainte des coûts réduits, d'améliorer le calcul de minorants de cette méthode. Des expérimentations sur les instances cb10.500 ont montré qu'elle donnait de meilleurs résultats que l'une des heuristiques les plus performantes connues. Bien que l'algorithme proposé dans ce chapitre soit relativement éloigné de l'implémentation originale de Resolution search proposée par Chvátal, il montre que cette méthode constitue un schéma d'exploration intéressant pour la résolution de problèmes d'optimisation linéaire à variables binaires.

Tab. 4.5 – Amélioration du calcul de minorants pour les instances cb10.500

Instance	Wilbaut, Hanafi [72]		RSBB		RSBBC	
	BI	t (s)	BI	t (s)	BI	t (s)
cb10.500_0	117809	3086	117809	141	**117809**	**13**
cb10.500_1	119217	2068	119222	5180	**119222**	**4059**
cb10.500_2	**119215**	**5028**	119211	410	119211	11
cb10.500_3	118801	3400	118813	860	**118813**	**26**
cb10.500_4	116514	241	116514	544	**116514**	**36**
cb10.500_5	119490	3130	119504	8375	**119504**	**4556**
cb10.500_6	**119794**	**2810**	119827	9862	119790	103
cb10.500_7	**118333**	**6146**	118323	1	118323	3
cb10.500_8	**117781**	**2200**	117779	10	117779	8
cb10.500_9	119251	4282	119251	11423	**119251**	**147**
cb10.500_10	217377	9	217377	18	**217377**	**4**
cb10.500_11	219077	1727	219077	2083	**219077**	**123**
cb10.500_12	217847	428	217847	18	**217847**	**7**
cb10.500_13	216868	1032	216868	67	**216868**	**7**
cb10.500_14	213853	1707	213861	1127	**213861**	**1067**
cb10.500_15	215086	1317	215086	123	**215086**	**14**
cb10.500_16	217940	4606	217915	1973	**217940**	**2944**
cb10.500_17	**219990**	**5729**	219984	73	219984	21
cb10.500_18	214375	329	214375	196	**214382**	**6014**
cb10.500_19	220886	3174	220899	744	**220899**	**659**
cb10.500_20	304387	5118	304363	40	**304387**	**4041**
cb10.500_21	302379	1462	302379	58	**302379**	**17**
cb10.500_22	302416	85	302416	12	**302416**	**22**
cb10.500_23	300784	2004	300784	3345	**300784**	**40**
cb10.500_24	304374	1561	304374	683	**304374**	**64**
cb10.500_25	301836	5314	301796	13	**301836**	**2648**
cb10.500_26	304952	326	304952	169	**304952**	**26**
cb10.500_27	296472	785	296478	4122	**296478**	**1168**
cb10.500_28	301357	1015	301353	3141	**301359**	**1867**
cb10.500_29	307089	839	307089	4324	**307089**	**17**

Chapitre 5

Application de Resolution search au problème de planification de techniciens et d'interventions pour les télécommunications

Nous présentons, dans ce chapitre, un travail réalisé dans le cadre du $5^{\text{ème}}$ challenge de la Société Française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF)[1]. Le sujet, proposé par *France Télécom*, concernait la planification de techniciens et d'interventions dans le domaine des télécommunications. Nous détaillons, dans une première partie, l'algorithme de recherche de bornes supérieures que nous avons présenté pour ce challenge et nous exposons ensuite une étude réalisée a posteriori sur l'application de Resolution search à la résolution d'un sous–problème du problème général.

L'objectif du problème que l'on note TIST (pour *Technicians and Interventions Scheduling Problem for Telecommunications*) est de concevoir des ordonnancements de techniciens et d'interventions dans le but d'aider les décideurs de *France Télécom* dans la réalisation de leurs plannings. Chaque jour, des équipes de techniciens doivent être formées afin d'effectuer différentes tâches à différents lieux géographiques. L'accroissement du nombre de clients et la diversification des services dus au développement du haut débit comme la voix sur ip ou la télévision par Internet rendent la réalisation des plannings de plus en plus complexe.

Les interventions sont caractérisées par des critères spécifiques comme une priorité de planification et une durée d'exécution, et certaines interventions doivent être réalisées avant d'autres. Les interventions sont également composées de différentes tâches qui nécessitent un certain nombre de techniciens d'un certain niveau de compétence dans un domaine donné. Les techniciens sont spécialisés dans différents domaines avec différents niveaux de compétence et chaque technicien a une liste de jours d'indisponibilité. D'autre part, chaque

[1] `http ://www.g-scop.fr/ChallengeROADEF2007/` ou `http ://www.roadef.org/`

intervention a un coût donné si elle est sous–traitée par une entreprise extérieure. Le coût total de ces interventions sous–traitées ne peut dépasser un certain budget. Le TIST consiste à ordonnancer un ensemble d'interventions en minimisant une fonction scalaire qui attribue une pénalité plus grande aux interventions les plus prioritaires. Ce problème a deux aspects combinatoires distincts : l'ordonnancement des interventions (qui dépend des contraintes de précédence) et la construction des équipes de techniciens (qui dépend du jour courant).

Dans une première partie, nous décrivons l'algorithme de résolution que nous avons présenté lors du challenge ROADEF'07. Cet algorithme, basé sur la métaheuristique GRASP (*Greedy Randomized Adaptive Search Procedure* [**?**]), donne des résultats prometteurs et nous a permis d'être classé $4^{ème}$ sur 35 équipes participantes lors du classement final. Néanmoins, l'un des défauts majeurs de cet algorithme concerne le choix des interventions à sous–traiter. Ce choix est effectué par une heuristique simple consistant à choisir dès le départ un ensemble d'interventions à sous–traiter et à ne plus considérer ces interventions durant la recherche. Un mauvais choix de départ peut donc pénaliser l'efficacité de l'algorithme durant tout le processus de recherche. Nous avons réalisé une étude a posteriori sur ce problème dans le but d'améliorer les résultats précédemment obtenus. Nous présentons, dans la deuxième partie de ce chapitre, les différentes voies qui ont été explorées pour cette étude et montrons en particulier que l'utilisation de Resolution search permet d'obtenir de meilleurs résultats que les autres méthodes testées sur la résolution de ce sous–problème. Une partie du travail décrit ici a fait l'objet de publications [42, 8].

5.1 Description du problème

Dans cette section, nous commençons par décrire le problème de manière informelle, puis nous présentons les différentes notations utilisées pour les données du problème. Nous concluons par une formulation mathématique du TIST dans laquelle les interventions sous–traitées ne sont pas prises en compte.

5.1.1 Description générale

Le problème traite d'interventions devant être affectées à des équipes de techniciens. Les techniciens sont caractérisés par leurs jours de congés et leurs niveaux de compétence, et les interventions par leur priorité, leur temps d'exécution, leur liste de prédécesseurs (interventions qui doivent être traitées avant l'intervention) et le nombre de techniciens requis pour chaque niveau et domaine de compétence. L'objectif consiste à construire des équipes de techniciens pour chaque journée et à affecter des interventions à ces équipes en respectant toutes les contraintes de la planification et en minimisant la fonction objective

$$28t_1 + 14t_2 + 4t_3 + t_4$$

où t_k est la date de fin de la dernière intervention de priorité k pour $k = 1, 2, 3$ et t_4 est la date de fin de l'ordonnancement.

Un ordonnancement doit satisfaire une liste de contraintes pour l'affectation des techniciens et pour l'affectation des interventions. Nous considérons que chaque journée de travail est dans l'intervalle de temps $[0, H_{\max}]$ et qu'il est impératif de respecter cet intervalle. En conséquence, une intervention ne peut être réalisée avant la date 0 ou après la date $H_{\max}$ et ne peut être réalisée sur plusieurs jours.

Une intervention doit être réalisée par une seule équipe à une unique date, plusieurs équipes ne peuvent se partager la même intervention et aucune intervention ne peut être affectée à un technicien en repos. Une contrainte forte est que les équipes ne peuvent changer durant la journée, ce qui implique que chaque technicien appartient au plus à une seule équipe chaque jour. Cette contrainte est due au nombre limité de voitures disponibles et au temps que cela prendrait de rapatrier les voitures pour former de nouvelles équipes.

Une équipe doit satisfaire les demandes de chaque intervention qui lui est assignée. Ainsi, pour chaque intervention, nous devons affecter suffisamment de techniciens qualifiés pour satisfaire toutes les demandes. Par exemple, une intervention qui nécessite un technicien de niveau 2 dans le domaine d1 peut être réalisée par un technicien de niveau 2, 3 ou 4 dans le domaine d1, mais ne peut être réalisée par deux techniciens de niveau 1 dans le domaine d1. Le nombre requis de techniciens d'un certain niveau pour une intervention est cumulable puisqu'un technicien d'un niveau donné est aussi qualifié pour tous les niveaux inférieurs pour le même domaine de compétence.

Finalement, il est possible de sous–traiter certaines interventions à une entreprise externe. Chaque intervention a un coût spécifique dans le cas où elle serait sous–traitée et le coût total de la sous–traitance ne peut excéder un budget donné. Notons que le modèle mathématique que nous exposons en section 5.1.2 ne prend pas en compte les interventions sous–traitées. En effet, cette partie du problème est réalisée dans une phase de prétraitement décrite en section 5.2.1.

La méthode GRASP consiste à alterner itérativement une phase de construction et une phase d'amélioration. La phase de construction génère une solution réalisable en insérant des interventions suivant un critère d'insertion, le voisinage de cette solution est ensuite exploré avec un algorithme de recherche locale dans la phase d'amélioration jusqu'à ce qu'un minimum local soit identifié. Un des inconvénients de l'implémentation classique de la méthode GRASP est qu'elle ne prend pas en compte l'information fournie par les solutions précédemment explorées car chaque itération est indépendante de l'autre. Nous proposons une implémentation intégrant l'apprentissage[2] dans le but de diriger la recherche vers des solutions de qualité. En effet, à chaque itération, les critères d'insertion utilisés pour construire une solution réalisable sont mis à jour en prenant en compte les explorations passées.

[2]Une bibliographie sur d'autres versions de GRASP intégrant l'apprentissage peut être trouvée dans l'article de Pitsoulis et Resende [57].

Notre approche est centrée sur trois phases principales : une phase de prétraitement qui sélectionne un sous–ensemble d'interventions à sous–traiter ; une phase d'initialisation qui cherche à identifier de bons critères d'insertion pour la phase de construction ; et une phase de recherche qui utilise GRASP pour trouver la meilleure solution possible.

5.1.2 Notations et modèle mathématique

Dans cette section, nous introduisons un ensemble de notations ainsi que le modèle mathématique du problème.

Les constantes du problème sont les suivantes :

- $H_{\max}$ est la durée de temps de chaque jour ($H_{\max} = 120$ dans le sujet).
- $T(I)$ est le temps d'exécution de l'intervention I.
- $cost(I)$ est le coût de l'intervention I.
- A est le budget alloué pour les interventions sous–traitées.
- $P(t, j)$ est égale à 1 si le technicien t travaille le jour j, 0 sinon.
- $C(t, i)$ est le niveau de compétence du technicien t dans le domaine i.
- $R(I, i, n)$ est le nombre de techniciens requis de niveau n dans le domaine i pour traiter l'intervention I.
- $Pred(I)$ est la liste des prédecesseurs de I.
- $Succ(I)$ est la liste des successeurs de I.

Les variables sont les suivantes :

- $s(I)$ est la date de début de l'intervention I.
- $e(t, j)$ est le numéro de l'équipe à laquelle est rattaché le technicien t pour le jour j. L'équipe 0 est une équipe spécifique composée des techniciens ne travaillant pas ce jour.
- $d(I)$ est le jour où l'intervention I est planifiée.

Une intervention qui nécessite, pour le domaine i, au moins un technicien de niveau 3 et un technicien de niveau 2 aura ses demandes notées : $R(I, i, 1) = 2$, $R(I, i, 2) = 2$, $R(I, i, 3) = 1$ et $R(I, i, 4) = 0$.

Pour le modèle mathématique, nous utilisons les constantes suivantes :

- $Pr(k, I)$ égale 1 si la priorité de l'intervention I est k et 0 sinon.
- $\mathcal{P}(I_1, I_2)$ égale 1 si l'intervention I_1 est un prédécesseur de l'intervention I_2 et 0 sinon.

et les variables suivantes :

- $x(I, j, h, \epsilon)$ égale 1 si l'équipe ϵ travaille sur l'intervention I le jour j à la date de début h et 0 sinon.
- $y(j, \epsilon, t)$ égale 1 si le technicien t est dans l'équipe ϵ le jour j et 0 sinon.
- t_k, $k = 1, 2, 3$ est la date de fin de la dernière intervention de priorité k.
- t_4 est la date de fin de l'ordonnancement.

Ce problème, noté (P'), peut être modélisé de la manière suivante :

$$\text{Minimiser} \quad 28t_1 + 14t_2 + 4t_3 + t_4$$

$$\text{sujet à} \quad \sum_{j,h,\epsilon} x(I,j,h,\epsilon) = 1, \qquad \forall I, \tag{5.1}$$

$$y(j,0,t) = 1 - P(t,j), \qquad \forall j,t, \tag{5.2}$$

$$\sum_{\epsilon} y(j,\epsilon,t) = 1, \qquad \forall j,t, \tag{5.3}$$

$$x(I,j,h,0) = 0, \qquad \forall I,j,h, \tag{5.4}$$

$$\sum_{h_1=\max(h_2-T(I_1)+1,0)}^{\min(h_2+T(I_2)-1,H_{\max})} x(I_1,j,h_1,\epsilon) + x(I_2,j,h_2,\epsilon) \leq 1, \quad \forall I_1,I_2,h_2,j,\epsilon, \tag{5.5}$$

$$\sum_{j,h,\epsilon} (jH_{\max}+h)\,(x(I_1,j,h,\epsilon) - x(I_2,j,h,\epsilon)) + T(I_1)x(I_1,j,h,\epsilon) \leq 0,$$

$$\forall I_1,I_2 \mid \mathcal{P}(I_1,I_2) = 1, \tag{5.6}$$

$$x(I,j,h,\epsilon) = 0 \qquad \forall I,j,h,\epsilon \mid h + T(I) > H_{\max}, \tag{5.7}$$

$$\sum_{h} R(I,i,n)x(I,j,h,\epsilon) \leq \sum_{t|C(t,i)\geq n} y(j,\epsilon,t), \qquad \forall I,i,n,\epsilon,j, \tag{5.8}$$

$$\sum_{j,h,\epsilon} (jH_{\max}+h+T(I))\,Pr(k,I)x(I,j,h,\epsilon) \leq t_k, \quad \forall I, k = 1,2,3, \tag{5.9}$$

$$\sum_{j,h,\epsilon} (jH_{\max}+h+T(I))\,x(I,j,h,\epsilon) \leq t_4, \qquad \forall I. \tag{5.10}$$

La contrainte (5.1) assure que chaque intervention est traitée par une seule équipe à une journée et à une date fixée. La contrainte (5.2) garantit que si un technicien t ne travaille pas le jour j, il est dans l'équipe 0. La contrainte (5.3) spécifie qu'un technicien appartient à une seule équipe chaque jour. La contrainte (5.4) assure qu'aucune intervention n'est réalisée par l'équipe 0. La contrainte (5.5) certifie que deux interventions réalisées le même jour par la même équipe sont effectuées à des dates différentes. La contrainte (5.6) spécifie que tous les prédécesseurs d'une intervention donnée doivent être réalisés avant la date de début de l'intervention. La contrainte (5.7) assure que chaque jour a une durée limite de $H_{\max}$, nombre maximal de portions de temps par jour. La contrainte (5.8) spécifie qu'une équipe qui travaille sur l'intervention I satisfait les demandes en nombre de techniciens par niveau de compétence. Finalement, la contrainte (5.9) spécifie que t_k est la date de fin de la dernière intervention de priorité k, $k = 1,2,3$ et la contrainte (5.10) spécifie que t_4 est la date de fin de l'ordonnancement.

5.2 Méthode de résolution

Dans cette section, nous détaillons les trois phases principales de notre approche : (i) la phase de prétraitement qui consiste à sélectionner un sous–ensemble d'interventions à sous–traiter ; (ii) la phase d'initialisation qui cherche à identifier de bons critères d'insertions pour la construction des solutions ; et (iii) la phase de recherche qui s'appuie sur la métaheuristique GRASP pour trouver la meilleure solution possible. La terminologie GRASP se réfère à une classe de procédures dans laquelle une heuristique gloutonne et une technique de recherche locale sont employées. La métaheuristique GRASP a été appliquée à de nombreux problèmes d'optimisation combinatoire comme les problèmes d'ordonnancement [75], les problèmes de routage [1], les problèmes de graphes [59], les problèmes d'affectation [23], les problèmes de planification [75] etc. On peut se référer à [**?**] pour une bibliographie plus complète sur le sujet.

La méthode GRASP consiste à répéter la procédure suivante jusqu'à satisfaire un critère d'arrêt (nombre maximum d'itérations, temps CPU fixé, niveau de qualité de la solution...) :

1. Générer une solution réalisable avec un algorithme glouton randomisé.
2. Appliquer un algorithme de recherche locale à la solution précédente.
3. Mettre à jour la meilleure solution.

5.2.1 Heuristique de choix des interventions à sous–traiter

Dans le contexte du challenge, nous devions trouver une méthode efficace dans un temps limité. Pour cela, nous avons opté pour une méthode heuristique qui consiste à choisir un ensemble d'interventions à sous–traiter au début du processus. Une fois cet ensemble déterminé, les interventions sous–traitées ne sont plus prises en compte durant le processus de recherche. Cette heuristique est fondée sur un critère d'insertion lié à un poids spécifique attribué à chaque intervention. Ce poids $\omega(I)$ est établi à partir d'une borne supérieure du nombre minimum de techniciens nécessaires à la réalisation de l'intervention I ($\texttt{mintec}(I)$), et de la durée de celle-ci ($T(I)$). Il est égal au produit de ces deux valeurs : $\omega(I) = \texttt{mintec}(I) \times T(I)$.

Soit Ω_t l'ensemble des indices des variables représentant les techniciens et $x \in \{0,1\}^{|\Omega_t|}$ un vecteur de variables de décision. Dans un premier temps, la valeur $\texttt{mintec}(I)$ est calculée en résolvant de manière heuristique le programme linéaire en nombres entiers suivant :

$$\begin{aligned} \text{Minimiser} \quad & \sum_{t \in \Omega_\text{t}} x_t \\ \text{sujet à} \quad & \sum_{t|C(t,I) \geq n, t \in \Omega_\text{t}} x_t \geq R(I,i,n), \quad \forall i,n, \\ & x_t \in \{0,1\}, \quad t \in \Omega_\text{t}. \end{aligned}$$

L'heuristique utilisée pour résoudre ce problème est décrite par l'algorithme 5.1.

Algorithme 5.1 – Calcul de la valeur `mintec`(I) pour une intervention I

```
mintec(I)
{
   ϵ := équipe vide ;
   T := sous-ensemble de techniciens satisfaisant au moins une demande de I ;
   for(chaque technicien t dans T) {
      sk(t,I) := nombre de niveaux de compétence de t satisfaisant
      les demandes de I pour tous les domaines ;
   }
   réarranger T selon l'ordre décroissant des valeurs sk(t,I) ;
   while(ϵ ne satisfait pas les demandes de I) {
      ajouter un nouveau technicien t ∈ T à ϵ ;
      mettre à jour T := T − {t} ;
   }
   for(chaque paire de techniciens {t,t'} dans ϵ)
      for(chaque technicien t'' de T)
         if({t,t'} peut être remplacé par {t''} pour satisfaire les compétences)
            remplacer {t,t'} par {t''} dans ϵ ;
            return nombre de techniciens dans ϵ ;
      }
   }
}
```

Soit $\omega(I)$ l'ensemble des indices des interventions et $x \in \{0,1\}^{|\omega(I)|}$ le vecteur de variables de décision tel que $x_I = 1$ si l'intervention I est sous–traitée et 0 sinon. Nous devons trouver, ensuite, un sous–ensemble d'interventions Γ à sous–traiter tel que $\sum_{x_I \in \Gamma} w_I x_I$ soit maximal et le coût total ne dépasse pas le budget total disponible A, ce qui correspond à un problème de sac à dos avec contraintes de précédence [44]. Ce problème, noté PCKP (pour *Precedence Constraint Knapsack Problem*), peut être formulé de la manière suivante :

$$
\begin{aligned}
\text{(PCKP) Maximiser} \quad & \sum_{I \in \Omega_\mathrm{I}} w_I x_I \\
\text{sujet à} \quad & \sum_{I \in \Omega_\mathrm{I}} cost(I) \cdot x_I \leq A, \\
& x_I \leq x_{I'}, \quad \forall I, I' \in \Omega_\mathrm{I} \mid \mathcal{P}(I, I') = 1, \\
& x_I \in \{0,1\}, \quad I \in \Omega_\mathrm{I}.
\end{aligned}
$$

Il est résolu par un algorithme glouton qui sélectionne les interventions de ratio $\omega(I)/cost(I)$ maximal, pour lesquelles tous les successeurs sont sous–traités, tant que le coût total ne dépasse pas le budget disponible.

Bien que ce choix de gestion des interventions à sous–traiter ne soit pas optimal, l'expérimentation a montré qu'il fournit de meilleurs résultats que d'autres stratégies testées telles que :

- Fixer le maximum de variables pour réduire le problème. Ce qui correspond à affecter un poids $w_I = 1$ pour toutes les interventions.
- Prendre en compte la priorité des interventions, c'est–à–dire fixer $w_I = 28$ si I est de priorité 1, $w_I = 14$ si I est de priorité 2 etc.

5.2.2 Phase de construction

Dans une implémentation classique de la métaheuristique GRASP, la phase de construction consiste à déterminer un ensemble d'éléments candidats pouvant être ajoutés à la solution partielle courante tout en maintenant la réalisabilité. La sélection de l'élément à incorporer est déterminée selon un poids : les éléments correspondant aux poids les plus forts sont insérés en priorité. Les poids sont donnés par un algorithme glouton qui évalue l'augmentation de la fonction objective qu'engendre l'insertion des candidats. Dans notre cas, ils sont tout d'abord initialisés à une valeur donnée (liée au coût de la priorité de l'intervention dans la fonction objective) et mis à jour à chaque itération en prenant en compte les solutions précédemment rencontrées. La méthodologie GRASP classique considère une liste restrictive de candidats (RCL pour *restricted candidate list*) composée des α % des candidats ayant les poids les plus élevés, $\alpha \in [0, 100]$. Les candidats sont alors choisis aléatoirement dans la RCL afin de les insérer dans la solution partielle courante. Dans notre approche, la RCL est composée de *toutes* les interventions, ce qui correspond à fixer la variable α à la valeur 100. Les interventions sont insérées selon l'ordre décroissant des poids et sont choisies aléatoirement en cas d'égalité.

Sélection d'un candidat

Initialement, le poids d'un candidat est fixé à la valeur du coefficient de sa priorité dans la fonction objective et nous fixons arbitrairement un poids d'une valeur de 1 pour les interventions de priorité 4. Ainsi, les candidats de priorité 1 (respectivement 2, 3) ont un poids de 28 (resp. 14 ,4) et les candidats de priorité 4 ont un poids de 1. L'algorithme glouton sélectionne le candidat qui a la plus haute valeur de poids. Lorsque deux candidats ont le même poids, le choix est fait aléatoirement. D'autre part, un candidat ne peut être sélectionné si au moins l'un de ses prédécesseurs ne l'est pas.

L'algorithme glouton cherche à insérer un candidat selon les trois critères suivants : (1) le jour le plus tôt ; (2a) l'équipe qui nécessite le moins de techniciens supplémentaires pour traiter l'intervention ; (2b) la date de début au plus tôt. Le processus est répété jusqu'à ce que tous les candidats soient ordonnancés. Nous décrivons maintenant comment l'algorithme tient compte de ces trois critères.

Recherche du jour au plus tôt

La première étape consiste à calculer la date de départ au plus tôt de l'intervention I. Elle correspond à un couple de valeurs *jour* et *heure* notées $d(I)$ et $s(I)$. Pour cela, nous cherchons la date de fin au plus tard parmi tous les prédécesseurs de I, $s(I_{max}) + T(I_{max})$ telle que $I_{max} \in Pred(I)$ et I_{max} est insérée le jour d_{max}. Si $s(I_{max})+T(I_{max})+T(I) > H_{\max}$ alors $d(I) = d_{max} + 1$ sinon $d(I) = d_{max}$. Dans tous les cas $s(I) = s(I_{max}) + T(I_{max})$.

Calcul du nombre minimum de techniciens requis pour un candidat

Les critères (2a) et (2b) dépendent des techniciens disponibles et des équipes existantes le jour $d(I)$. Afin de respecter le critère (2a), le nombre de techniciens nécessaire à la construction d'une nouvelle équipe doit être connu. L'algorithme vérifie que les niveaux de compétence requis par l'intervention I sont satisfaits par une équipe donnée ϵ. Si c'est le cas, il n'est pas nécessaire d'ajouter un technicien à l'équipe ϵ. Dans le cas contraire, le nombre minimum de techniciens à ajouter à ϵ est déterminé de manière heuristique à partir des techniciens disponibles le jour $d(I)$. Nous notons $techs^{\epsilon}(I)$ le nombre de techniciens nécessaires pour assigner I à l'équipe ϵ ($\epsilon = 0$ si une nouvelle équipe doit être créée). Il reste à calculer la date de début au plus tôt de I pour ces équipes.

Calcul de la date de début au plus tôt

L'étape suivante de l'algorithme consiste à calculer la date au plus tôt $s^{\epsilon}(I)$ de planification de l'intervention I avec l'équipe ϵ. Dans le cas d'une nouvelle équipe, il s'agit simplement de la valeur $s(I)$. Sinon, pour chacune des équipes retenues, on essaie d'insérer I au plus tôt dans leur planning.

Choix entre (2a) et (2b)

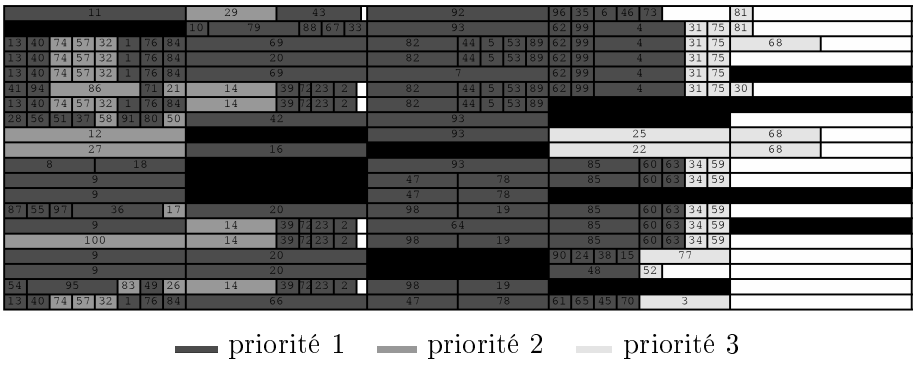
L'ordre des critères (2a) et (2b) dépend de la valeur de $d(I)$ obtenue précédemment. Il dépend également de la date de fin de la dernière intervention de même priorité que I pour les priorités 1, 2, 3 et de la date de fin totale pour la priorité 4 (i.e. t_i, où i désigne la priorité du candidat I). Si $d(I) \times H_{\max} + s(I) + T(I) < t_i$ alors la condition (2a) est considérée avant la condition (2b). Sinon, la condition (2b) a la priorité. En effet, si l'insertion de I ne provoque pas d'augmentation de la date de fin de la dernière intervention de même priorité que I, alors elle n'influe pas sur la valeur de la fonction objective. Dans ce cas, nous cherchons à minimiser le nombre de techniciens à ajouter avant de minimiser la valeur de $s(I)$.

En résumé, pour favoriser la condition (2a), l'algorithme choisit l'équipe ϵ avec la valeur minimale $techs^{\epsilon}(I)$ correspondante. Si plusieurs équipes ont la même valeur, l'équipe choisie est celle pour laquelle la valeur $s^{\epsilon}(I)$ est minimale. Pour favoriser la condition (2b), l'algorithme choisit l'équipe ϵ avec la valeur minimale $s^{\epsilon}(I)$ et celle ayant la valeur minimale $techs^{\epsilon}(I)$ s'il existe au moins deux équipes avec la même valeur $s^{\epsilon}(I)$.

Utilisation d'une permutation des poids dans l'algorithme glouton

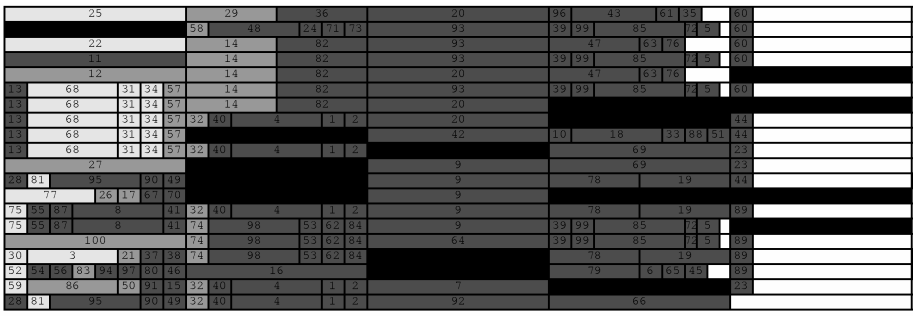
Des expériences préliminaires nous ont montré que le critère de choix des interventions pour l'algorithme glouton n'est pas trivial et, en particulier, qu'il ne correspond pas toujours à l'ordre des coefficients des priorités dans la fonction objective. Notons $w(I)$ le poids de l'intervention I. Supposons que $w(I)$ soit égal au coefficient de la priorité de I dans la fonction objective. Cela revient à choisir les interventions de haute priorité d'abord. La figure 5.1 représente une solution générée par l'algorithme glouton dans ces conditions. Les poids des interventions sont : 28 pour les interventions de priorité 1 ; 14 pour les interventions de priorité 2 ; 4 pour les interventions de priorité 3 ; 1 pour celles de priorité 4. Dans cette figure, chaque ligne représente un technicien : le premier technicien est représenté par la ligne du haut et le dernier technicien par la ligne du bas. Chaque rectangle noir correspond à un jour de congés et chaque ligne verticale correspond à la fin d'une journée. La valeur de la fonction objective de cette solution est 17820.

Fig. 5.1 – Solution avec un objectif de 17820 pour l'instance 8 de l'ensemble data-setA



Il est possible d'affecter les poids aux interventions de manière différente. Supposons que les interventions de priorité 4 aient un poids de 28, celles de priorité 3 un poids de 14, celles de priorité 1 un poids de 4 et celles de priorité 2 un poids de 1. Cette affectation correspond à utiliser la permutation (4,3,1,2) des poids. La figure 5.2 donne une solution générée avec l'algorithme glouton en utilisant ces poids. La valeur de l'objectif de cette solution est 17355. Notons que les poids des interventions ne sont pas utilisés pour évaluer la solution mais uniquement pour guider l'algorithme glouton.

FIG. 5.2 – Solution avec un objectif de 17355 pour l'instance 8 de l'ensemble data-setA



Cet exemple montre que, pour cette instance, il est préférable de fixer un poids fort pour les interventions de priorité 3 et d'utiliser la permutation (4,3,1,2) des poids associés aux priorités. Nous précisons que les permutations (4,3,1,2), (3,2,1,4), (4,3,2,1) et (4,2,1,3) sont équivalentes pour l'algorithme glouton dans le cas de figure où il n'y a pas d'intervention de priorité 4.

Mise à jour des poids

Un des inconvénients potentiels de l'architecture GRASP standard est qu'elle ne prend pas en compte les solutions précédemment visitées. Dans l'implémentation proposée, nous utilisons l'information fournie par les solutions précédentes pour diriger la recherche vers des zones potentiellement « prometteuses ». Ceci est réalisé, à chaque itération, en mettant à jour les poids des candidats en considérant les caractéristiques des solutions précédentes. Notons $w_p(I)$, le poids associé à l'intervention I selon la permutation p des poids associés aux priorités. Par exemple, supposons que $p = (3, 2, 1, 4)$, alors $w_p(I) = 28$ si la priorité de I est 3, $w_p(I) = 14$ si la priorité de I est 2, etc. À la fin de l'exécution de l'algorithme glouton, les poids des interventions sont mis à jour à partir de l'information fournie par la solution générée [65]. Cette mise à jour consiste à ajouter la valeur $w_p(I)$ aux dernières interventions de chaque priorité et à tous leurs prédécesseurs de manière à planifier ces interventions plus tôt à l'itération suivante. L'algorithme 5.2 illustre cette procédure.

Algorithme 5.2 – Phase de mise à jour

```
mise_a_jour_poids(p)
{
   for(chaque priorité prio) {
      I := dernière intervention planifiée de priorité prio ;
      w(I) = w(I) + w_p(I) ;
      for(chaque intervention J ∈ Pred(I))
         w(J) = w(J) + w_p(I) ;
   }
}
```

5.2.3 Phase d'initialisation

L'identification de la permutation qui conduit au meilleur comportement de l'algorithme glouton est réalisée par la procédure d'échantillonnage suivante :

- Appliquer plusieurs fois l'algorithme glouton pour chacune des 24 permutations des poids associés aux priorités. Trier les permutations selon la meilleure borne supérieure obtenue.
- Répéter la même procédure sur les 12 permutations qui donnent les meilleures valeurs.
- Répéter la même procédure sur les 6 permutations qui donnent les meilleures valeurs.

Lorsque cette phase est terminée, nous conservons les 2 meilleures permutations. L'algorithme GRASP utilisera ces deux permutations pour générer toutes les autres solutions : les poids des interventions évoluant depuis ces valeurs de départ.

5.2.4 Phase d'amélioration

Dans cette section, nous décrivons un algorithme de recherche locale qui explore le voisinage des solutions rencontrées en vue d'une amélioration.

Après avoir affecté les interventions aux équipes et déterminé l'ordre dans lequel les interventions sont traitées pour chaque équipe, nous vérifions la faisabilité de l'ordonnancement. De plus, les dates de début optimales des interventions sont déterminées facilement, dans ce cas, puisque le graphe représentant l'ordre d'exécution des interventions avec les contraintes de précédence est un arbre acyclique direct pondéré [16].

Nous proposons deux algorithmes de recherche locale, que nous appelons : phase *critical path* et phase *packing*. Nous utilisons un mouvement d'échange et un mouvement d'insertion, et ne considérons que les mouvements réalisables.

Lors de la recherche locale, nous maintenons le nombre minimal de techniciens affectés pour les interventions planifiées. Ainsi, les techniciens disponibles durant la journée appartiennent soit à une équipe vide pour laquelle aucune intervention n'est assignée, soit aux équipes pour lesquelles le nombre de techniciens est minimal pour les interventions qui leur sont assignées.

Une opération d'échange consiste à intervertir l'affectation et l'ordre de deux interventions et une opération d'insertion supprime une intervention et l'insère à une autre position temporelle.

Phase *critical path*

Le but de la phase *critical path* est de réduire les dates de fin de chaque priorité et la date de fin de l'ordonnancement global (i.e., t_1, t_2, t_3 et t_4) simultanément.

Un chemin critique pour une priorité est défini comme étant une séquence maximale $(I_1, I_2, \ldots, I_l)$ d'interventions telles que l'intervention I_l donne la date de fin de la priorité considérée, et chaque intervention consécutive I_k et I_{k+1} $(k = 1, \ldots, l-1)$ satisfait

$$d(I_k) = d(I_{k+1}) \text{ et } s(I_k) + T(I_k) = s(I_{k+1})$$

ou

$$d(I_k) + 1 = d(I_{k+1}),\ s(I_{k+1}) = 0 \text{ et } s(I_k) + T(I_k) + T(I_{k+1}) > H_{\max}.$$

L'intervention I_{k+1} ne peut être insérée si l'intervention I_k n'est pas insérée plus tôt. Par définition d'un chemin critique, l'intervention I_1 doit être séquencée plus tôt si nous voulons réduire la date de fin de la priorité.

L'algorithme de recherche locale cherche un chemin critique pour chaque priorité et essaie de rompre ce chemin en insérant l'intervention I_1 au plus tôt.

Phase *packing*

Dans la phase *packing*, l'algorithme cherche à insérer les interventions de manière efficace sans dégrader l'objectif.

Nous considérons une mesure de l'efficacité pour l'équipe ϵ du jour j. Soit $J_\epsilon = \{I_1, I_2, \ldots, I_l\}$ les interventions qui sont affectées à l'équipe ϵ. Soit $N(J_\epsilon, i, n)$ le nombre maximum de techniciens requis pour le niveau n et le domaine i pour réaliser les interventions de J_ϵ (i.e., $N(J_\epsilon, i, n) = \max_{I \in J_\epsilon} R(I, i, n)$). Soit

$$W_{\text{skill}}(J_\epsilon) = \sum_{I \in J_\epsilon} \sum_{i,n} (N(J_\epsilon, i, n) - R(I, i, n)) T(I)$$

et

$$W_{\text{time}}(J_\epsilon) = H_{\max} - \sum_{I \in J_\epsilon} T(I),$$

qui représentent respectivement les niveaux de compétence « gaspillés » et le temps « gaspillé » pour l'équipe ϵ et les interventions J_ϵ. Nous estimons l'efficacité de l'affectation des interventions J_ϵ à l'équipe ϵ par la fonction

$$f(J_\epsilon) = W_{\text{skill}}(J_\epsilon) + \alpha W_{\text{time}}(J_\epsilon),$$

où α est fixé à une grande valeur pour prendre en compte en priorité le temps puis les niveaux de compétence.

Dans cette phase, la recherche locale estime la valeur d'une solution en sommant $f(J_\epsilon)$ pour toutes les équipes de tous les jours, et elle accepte une solution voisine pour un mouvement si la solution est réalisable et qu'elle n'augmente pas les dates de fin de chaque priorité.

5.2.5 Schéma général de l'approche de résolution

L'algorithme 5.3 résume les trois phases exposées dans les sections précédentes. L'heuristique de prétraitement qui sélectionne les interventions à sous–traiter est représentée par la fonction `glouton_sous_traités`. Cette fonction retourne un sous–problème dans lequel une partie des variables a été fixée (i.e. avec quelques interventions supprimées). La phase d'initialisation consistant à assigner des poids initiaux aux interventions est décrite par la fonction `initialisation_poids`. Cette procédure fournit $conf_1$ et $conf_2$, qui correspondent aux deux configurations initiales des poids utilisées dans la procédure GRASP. La phase GRASP consiste à répéter l'exécution de l'algorithme glouton avec les deux configurations des poids sélectionnées, puis à mettre à jour la mémoire et finalement à appliquer la recherche locale lorsque la meilleure solution est améliorée. Le processus s'arrête lorsque le temps cpu alloué est dépassé.

Algorithme 5.3 – Algorithme général

```
résoudre_TIST(PB,MAX_CPU)
{
   meilleure_solution = ∅ ;
   SPB = glouton_sous_traités(PB) ;
   (conf1, conf2) = initialisation_poids(SPB) ;
   while(MAX_CPU n'est pas atteint) {
      solution_1 = construction_gloutonne(SPB, conf1) ;
      solution_2 = construction_gloutonne(SPB, conf2) ;
      Mettre à jour la mémoire ;
      amélioration = mise_a_jour(solution_1,solution_2,meilleure_solution) ;
      if(amélioration = true)
         meilleure_solution = recherche_locale(SPB, meilleure_solution) ;
   }
}
```

5.2.6 Calcul d'une borne inférieure

Dans cette section, nous considérons une borne inférieure du problème P' où les interventions sous–traitées ont été supprimées, afin d'évaluer la performance de l'algorithme.

Pour calculer la borne inférieure de P' nous considérons huit problèmes relaxés avec une restriction sur les interventions choisies. Sur chacun de ces huit problèmes, nous calculons une borne inférieure de la date de fin de l'ordonnancement (appelée makespan). La borne inférieure de P' est finalement calculée en utilisant ces valeurs.

Nous considérons les problèmes suivants :

- $MSP(1)$ avec uniquement les interventions de priorité 1 et leurs prédécesseurs.
- $MSP(2)$ avec uniquement les interventions de priorité 2 et leurs prédécesseurs.
- $MSP(3)$ avec uniquement les interventions de priorité 3 et leurs prédécesseurs.

- $MSP(1,2)$ avec uniquement les interventions de priorité 1 et 2 et leurs prédécesseurs.
- $MSP(2,3)$ avec uniquement les interventions de priorité 2 et 3 et leurs prédécesseurs.
- $MSP(3,1)$ avec uniquement les interventions de priorité 3 et 1 et leurs prédécesseurs.
- $MSP(1,2,3)$ avec les interventions de priorité 1 et 2 et 3 et leurs prédécesseurs.
- $MSP(1,2,3,4)$ avec toutes les interventions.

Si l'on note T_1, T_2, T_3, $T_{1,2}$, $T_{2,3}$, $T_{3,1}$, $T_{1,2,3}$ et $T_{1,2,3,4}$ les bornes inférieures des makespans respectifs, la résolution du problème suivant fournit une borne inférieure pour P' :

$$
\begin{aligned}
\text{Minimiser} \quad & 28t_1 + 14t_2 + 4t_3 + t_4 \\
\text{sujet à} \quad & T_1 \leq t_1,\ T_2 \leq t_2,\ T_3 \leq t_3, \\
& T_{1,2} \leq \max\{t_1, t_2\},\ T_{2,3} \leq \max\{t_2, t_3\},\ T_{3,1} \leq \max\{t_3, t_1\}, \\
& T_{1,2,3} \leq \max\{t_1, t_2, t_3\}, \\
& T_{1,2,3,4} \leq t_4,
\end{aligned}
$$

où t_1, t_2 et t_3 sont respectivement les dates de fin des priorités 1, 2 et 3, et t_4 est la date de fin de l'ordonnancement global. Toute solution réalisable doit satisfaire les contraintes précédentes.

Nous proposons trois bornes inférieures pour chaque problème : la borne inférieure par *boîtes* qui est calculée de manière combinatoire ; la borne inférieure par *affectation* qui est obtenue par la résolution d'un programme linéaire qui est un sous–problème de P' ; et enfin la borne *triviale* qui est dérivée de conditions triviales du problème. Nous choisissons la meilleure borne inférieure parmi celles-ci et la renforçons par une procédure d'amélioration.

Borne inférieure par boîtes

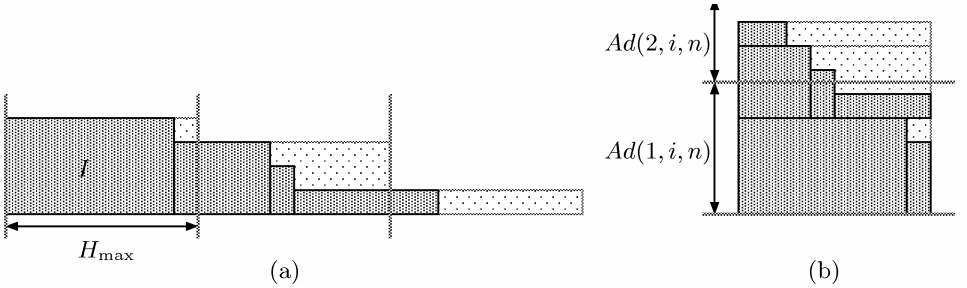
La borne inférieure par boîtes, qui est une borne inférieure sur le nombre de jours de l'ordonnancement, est calculée pour chaque domaine i et niveau de compétence n ; la plus grande valeur est conservée. La même méthode a été proposée par Lodi, Martello et Vigo [47] pour le problème *two-dimensional level packing problem*.

Pour chaque domaine i et niveau de compétence n, nous considérons un rectangle, associé à chaque intervention I, dont la hauteur est $R(I,i,n)$ et la largeur est $T(I)$.

Considérons tous les rectangles insérés bout à bout par hauteur décroissante comme dans la figure 5.3 (a).

Ces rectangles sont d'abord découpés en blocs de largeur $H_{\max}$ pour respecter la durée de travail d'une journée. Ces blocs ont une largeur égale à $H_{\max}$ et une hauteur égale au $R(I,i,n)$ de la première intervention I qui les constitue. Ils sont ensuite superposés pour en constituer un unique. Cette opération est illustrée par la figure 5.3 (b) où $Ad(j,i,n)$ est le nombre de techniciens qui peuvent travailler le jour j et dont le niveau de compétence dans le domaine i est n.

Fig. 5.3 – Illustration du calcul de la borne inférieure par boîtes



Nous calculons enfin $\mu^* = \min\{\mu \in Z \mid H \leq \sum_{j=1}^{\mu} Ad(j,i,n)\}$ qui représente le nombre minimum de jours nécessaires pour pouvoir couvrir la hauteur totale du bloc construit précédemment. μ^* est ainsi une borne inférieure du nombre de jours pour l'ordonnancement si l'on ne considère que le domaine i et le niveau n. La figure 5.3 (b) illustre une situation où $\mu^* = 2$.

Borne inférieure par affectation

Pour calculer la borne inférieure par affectation, nous estimons successivement le nombre de jours μ de l'ordonnancement en résolvant un programme linéaire correspondant à μ, et nous répétons le processus jusqu'à ce que l'estimation soit correcte.

Pour un nombre de jours μ donné (nous estimons alors que le makespan (date de fin de l'ordonnancement) M est dans $[\mu H_{\max}, (\mu+1)H_{\max}]$), le temps de travail disponible $U_\mu(t,M)$ jusqu'au temps M pour chaque technicien t peut être calculé facilement (notons que $U_\mu(t,M)$ est représenté par $M - lH_{\max}$ pour $l \leq \mu$ ou 0).

Nous considérons alors le programme linéaire suivant ALP(μ) :

$$\text{Minimiser} \quad M \tag{5.11}$$

$$\text{sujet à} \quad \sum_{t|C(t,i)\geq n} x_{I,t} \geq R(I,i,n), \qquad \forall I, \forall i, \forall n \tag{5.12}$$

$$\sum_{I} T(I) x_{I,t} \leq U_\mu(t,M), \qquad \forall t, \tag{5.13}$$

$$0 \leq x_{I,t} \leq 1 \qquad \forall I, \forall t, \tag{5.14}$$

où $x_{I,t}$ représente l'affectation de l'intervention I au technicien t. L'ensemble des solutions réalisables de ALP(μ) est inclus dans celui de ALP($\mu+1$) par conséquent la valeur optimale de ce problème devient plus petite lorsque μ augmente.

Si le problème n'a pas de solutions réalisables ou si la valeur optimale M^* de ALP(μ) est supérieure à $(\mu+1)H_{\max}$, nous en déduisons que la borne inférieure est plus grande. Si la valeur optimale M^* de ALP(μ) est inférieure à $(\mu+1)H_{\max}$, nous en déduisons que la borne inférieure est plus petite. Le processus est répété jusqu'à ce que l'estimation réussisse et la dernière valeur M^* s'avère être la borne inférieure par affectation.

Borne inférieure triviale

La borne inférieure triviale est dérivée des conditions nécessaires suivantes : (1) le temps maximal d'exécution d'une intervention parmi toutes les interventions est une borne inférieure, (2) la somme des temps d'exécution d'une séquence d'interventions liées par des contraintes de précédence est une borne inférieure. La valeur maximale de toutes ces séquences peut être calculée en temps linéaire et donne une borne inférieure du problème.

Renforcement de la borne

Sachant que le makespan est une combinaison de $T(I)$ et $H_{\max}$, il est possible de renforcer la borne inférieure courante. Pour cela, nous calculons tous les makespans possibles par un algorithme de programmation dynamique pour le jour associé à la borne inférieure, et nous prenons la valeur la plus petite supérieure ou égale à la borne inférieure donnée. Cette valeur renforce la borne inférieure trouvée.

5.2.7 Résultats préliminaires

Nous présentons dans cette section les résultats obtenus sur l'ensemble des données fournies par *France Télécom* pour ce challenge. Il y a trois ensembles de données disponibles, chaque ensemble contenant 10 instances avec un nombre différent d'interventions, de techniciens, de domaines de compétence et de niveaux de compétence. Le premier ensemble, appelé data-setA, ne considère pas le problème des interventions sous–traitées. Il contient des instances (notées Ai) ayant de 5 à 100 interventions, de 5 à 20 techniciens, de 3 à 5 domaines de compétence et de 2 à 4 niveaux de compétence. L'ensemble data-setB contient des instances (notées Bi) plus difficiles à résoudre qui prennent en compte le problème des interventions sous–traitées. Cet ensemble contient des instances ayant de 120 à 800 interventions, de 30 à 150 techniciens, de 4 à 40 domaines de compétence et de 3 à 5 niveaux de compétence. Finalement, l'ensemble data-setX (notées Xi) est l'ensemble d'instances à partir desquelles le classement des équipes participant au challenge a été réalisé. Il contient des instances ayant de 100 à 800 interventions, de 20 à 100 techniciens, de 6 à 20 domaines de compétence et de 3 à 7 niveaux de compétence.

Nous exposons, dans le tableau 5.1, les résultats officiels qui ont été publiés sur le site Web du challenge. L'ordinateur utilisé est un AMD avec un processeur de $1,8$ GHz et 1 GB de DDR-RAM. Le temps d'exécution est limité à 1200 secondes.

La description des données par colonne est la suivante : *Pb* : nom de l'instance. *int.* : nombre d'interventions. *tec.* : nombre de techniciens. *dom.* : nombre de domaines de compétence. *lev.* : nombre de niveaux de compétence. La colonne *GRASP* a trois valeurs : `BS` : la meilleure valeur de l'objectif trouvée par notre approche, `BI` : la valeur de la borne inférieure une fois les interventions sous–traitées choisies et *gap* : l'écart relatif entre la borne inférieure et la valeur de la fonction objective. La colonne *Meilleur objectif* a deux valeurs : `BS` : la meilleure valeur de l'objectif trouvée parmi tous les participants au challenge et *gap* : l'écart relatif entre cette meilleure valeur et la valeur que notre méthode a trouvée.

Tab. 5.1 – Résultats officiels obtenus lors du challenge

					GRASP			Meilleur objectif	
Pb	int.	tec.	dom.	lev.	BS	BI	gap.	BS	gap
A1	5	5	3	2	2340	2265	3.2	2340	0
A2	5	5	3	2	4755	4215	11.35	4755	0
A3	20	7	3	2	11880	11310	4.79	11880	0
A4	20	7	4	3	13452	10995	18.26	13452	0
A5	50	10	3	2	28845	26055	9.67	28845	0
A6	50	10	5	4	18870	17775	5.8	18795	0.39
A7	100	20	5	4	30840	27405	11.13	30540	0.97
A8	100	20	5	4	17355	16166	6.85	16920	2.50
A9	100	20	5	4	27692	25618	7.48	27692	0
A10	100	15	5	4	40020	35405	11.53	38296	4.3
						Moyenne	**9.01**		**0.81**
B1	200	20	4	4	43860	38385	12.48	34395	21.58
B2	300	30	5	3	20655	16605	19.6	15870	23.16
B3	400	40	4	4	20565	17460	15.09	16020	22.1
B4	400	30	40	3	26025	19035	26.85	25305	2.76
B5	500	50	7	4	120840	106290	12.04	89700	25.76
B6	500	30	8	3	34215	24450	28.54	27615	19.28
B7	500	100	10	5	35640	28470	20.11	33300	6.56
B8	800	150	10	4	33030	32820	0.63	33030	0
B9	120	60	5	5	29550	26310	10.96	28200	4.56
B10	120	40	5	5	34920	32790	6.09	34680	0.68
						Moyenne	**15.24**		**12.64**
X1	600	60	15	4	181575	140025	22.88	151140	16.76
X2	800	100	6	6	7260	6840	5.78	7260	0
X3	300	50	20	3	52680	49650	5.75	50040	5.01
X4	800	70	15	7	72860	59560	18.25	65400	10.23
X5	600	60	15	4	172500	126465	26.68	147000	14.78
X6	200	20	6	6	9480	6180	34.81	9480	0
X7	300	50	20	3	46680	45000	3.59	33240	28.79
X8	100	30	15	7	29070	20590	29.17	23640	18.67
X9	500	50	15	4	168420	101985	39.44	134760	19.98
X10	500	40	15	4	178560	99705	44.16	137040	23.25
						Moyenne	**23.05**		**13.74**

Un total de 17 équipes participaient à la phase finale du challenge. Notre algorithme a été placé en première position dans la catégorie *Junior* et en quatrième position dans le classement général.

Le tableau 5.1 montre que l'écart entre la solution trouvée par notre algorithme et la borne inférieure est important pour certaines instances de l'ensemble data-setX. Les expérimentations réalisées sur ces instances, après le challenge, ont montré que notre algorithme n'atteint jamais la phase d'amélioration par la recherche locale. Nous devons accélérer la phase 2 de notre approche, qui est en cause ici, soit en allégeant la procédure d'échantillonnage décrite en section 5.2.3 soit en la remplaçant par une méthode déterministe du calcul de la permutation optimale.

Un autre point mis en évidence par ces expérimentations est que le choix des interventions à sous–traiter est sous–optimal et pénalise notre algorithme GRASP. En effet, certaines valeurs de borne inférieure (calculées sur le problème résiduel ne contenant pas les interventions sous–traitées) sont supérieures à la meilleure solution trouvée parmi tous les participants (majorant du problème global). Nous allons voir dans la section suivante que l'amélioration de la procédure de choix de ces interventions permet d'améliorer les performances de notre algorithme.

5.3 Étude sur le choix des interventions à sous–traiter

Dans cette section, nous présentons une étude réalisée sur le problème du choix des interventions à sous–traiter. Rappelons que le problème consiste à sélectionner un sous–ensemble d'interventions de manière à ne pas dépasser le budget alloué et tel que tous les successeurs d'une intervention sous–traitée soient sous–traités. Une question majeure est l'évaluation de la pertinence de sous–traiter une intervention plutôt qu'une autre, et plus généralement de sous–traiter un sous–ensemble d'interventions plutôt qu'un autre. Nous présentons ici quelques pistes qui ont été étudiées sur ce problème et, en particulier, les expérimentations faites avec Resolution search qui nous ont permis d'améliorer les résultats précédemment obtenus.

5.3.1 Résolution exacte du sac à dos avec contraintes de précédence

Une première variante de l'algorithme initial consiste à remplacer la résolution heuristique du problème de sac à dos avec contraintes de précédence (cf. section 5.2.1) par une résolution exacte en utilisant CPLEX 9.2. Les résultats obtenus par cette version n'ont pas été globalement encourageants. Le tableau 5.2 présente un récapitulatif des résultats obtenus en comparaison avec la version heuristique utilisant l'algorithme glouton pour sélectionner les interventions à sous–traiter.

Dans ce tableau, les colonnes *BI* donnent les bornes inférieures obtenues avec le sous–ensemble d'interventions sous–traitées donné par l'heuristique gloutonne et celui donné par la résolution exacte du PCKP. Les colonnes *BS* donnent les bornes supérieure correspondantes. Les valeurs en gras indiquent les meilleures bornes supérieures trouvées parmi les deux méthodes.

Tab. 5.2 – Résultats obtenus par la résolution exacte du PCKP

Pb	Heuristique gloutonne		Méthode exacte		Pb	Heuristique gloutonne		Méthode exacte	
	BI	BS	BI	BS		BI	BS	BI	BS
B1	38385	**43860**	38835	44670	X1	140025	**181575**	140025	**181575**
B2	16605	20655	15930	**20640**	X2	6480	**7260**	6840	**7260**
B3	17460	**20565**	17535	21060	X3	49650	**52680**	50520	55740
B4	19035	**26025**	18690	26505	X4	59560	72860	59570	**71640**
B5	106290	120840	107670	**120630**	X5	126465	**172500**	126465	**172500**
B6	24450	**34215**	24390	34440	X6	6180	**9480**	7020	10740
B7	28470	**35640**	28080	35910	X7	45000	**46680**	45870	46920
B8	32820	**33030**	32820	**33030**	X8	20590	29070	20410	**27840**
B9	26310	**29550**	26310	**29550**	X9	101985	**168420**	103035	168660
B10	32790	**34920**	34470	36600	X10	99705	178560	100950	**176280**

Pour trois instances, les sous–ensembles d'interventions sous–traitées sont les mêmes (B9, X1 et X5). Nous pouvons voir qu'en moyenne, les résultats finaux obtenus sont moins bons du point de vue de la qualité de la borne supérieure finale. Ces résultats nous conduisent donc à chercher dans une autre direction.

5.3.2 Énumération et évaluation des sous–ensembles maximaux

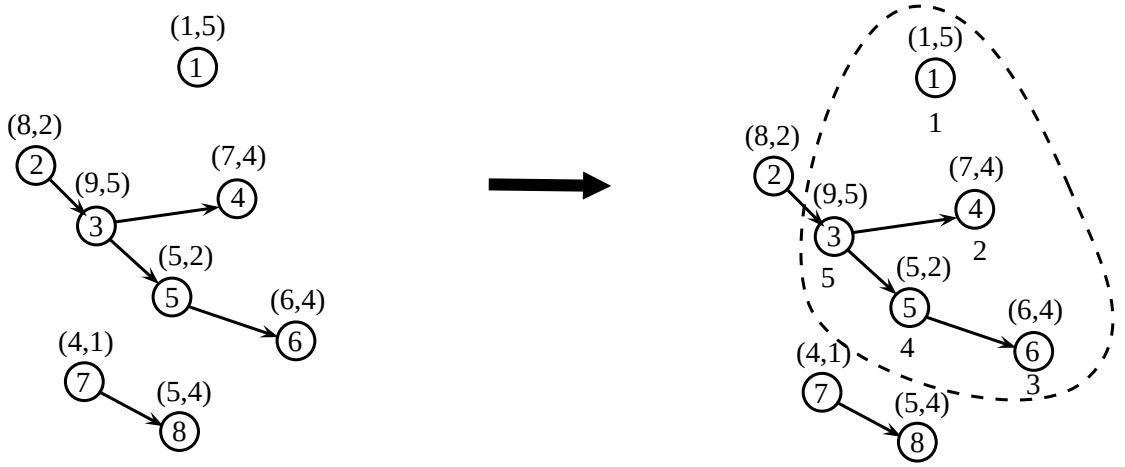
La seconde stratégie envisagée vise à énumérer une série de sous–ensembles maximaux d'interventions (au sens du budget) qui peuvent être sous–traitées, et les évaluer de manière à choisir le « meilleur ». L'évaluation d'un sous–ensemble d'interventions n'est pas triviale. L'approche envisagée consiste à utiliser le calcul de la borne inférieure (section 5.2.6) sur chaque sous–ensemble et à conserver le sous–ensemble générant la plus petite borne inférieure.

Après avoir déterminé les différentes composantes connexes du graphe formé par les interventions de l'instance, une stratégie récursive est mise en place pour parcourir toutes les interventions. Une intervention donnée peut être ajoutée au sous–ensemble courant si son coût n'excède pas le budget restant et si tous ses successeurs sont dans le sous–ensemble courant. L'algorithme commence par considérer les interventions « isolées » (i.e. sans prédécesseur ni successeur), puis il considère chaque composante connexe en commençant par les interventions n'ayant pas de successeurs. L'énumération se base également sur le critère d'insertion utilisé dans l'approche heuristique présentée en section 5.2.1, c'est–à–dire qu'elle va chercher à insérer en priorité les interventions ayant un plus grand ratio $\omega(I)$ défini par

$$\omega(I) = \frac{\texttt{mintec}(I) \times T(I)}{cost(I)}.$$

La figure 5.4 illustre la façon dont l'algorithme génère les sous–ensembles maximaux. On suppose dans cet exemple que le budget alloué est $A = 20$. Le graphe de gauche représente un ensemble d'interventions telles que les valeurs (*val1*,*val2*) sont respectivement le ratio $\omega(I)$ et le coût $cost(I)$, tandis que les flèches représentent les relations de précédence.

FIG. 5.4 – Principe de création des sous–ensembles maximaux par l'énumération



L'algorithme commence tout d'abord par insérer l'intervention 1 car elle n'a ni prédécesseur ni successeur. Il insère ensuite l'intervention 4, car parmi les interventions n'ayant pas de successeurs, elle correspond à celle ayant le plus grand ratio $\omega(I)$. L'intervention qui vient ensuite est l'intervention 6 car tous les successeurs de l'intervention 3 ne sont pas sous–traités et l'intervention 8 a une valeur $\omega(I)$ inférieure. Viennent ensuite les interventions 5 et 3 car l'intervention 2 ne peut être insérée à cause du manque de budget disponible. Lorsqu'un sous–ensemble maximal est identifié, il est évalué en calculant la borne inférieure sur les interventions non sous–traitées. L'ensemble donnant la plus petite borne inférieure constitue l'ensemble des interventions sous–traitées pour l'algorithme GRASP. Le tableau 5.3 expose les résultats obtenus en exécutant l'énumération durant 600 secondes.

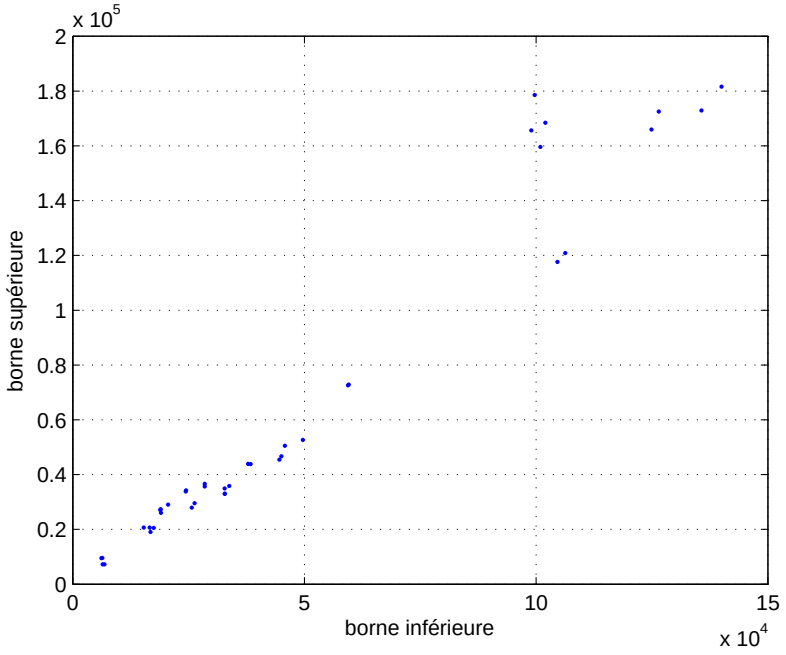
TAB. 5.3 – Résultats obtenus par l'énumération des sous–ensembles maximaux

Pb	Glouton		Enumération			Pb	Glouton		Enumération		
	BI	BS	BI	t(s)	BS		BI	BS	BI	t(s)	BS
B1	38385	**43860**	**37845**	401	43890	X1	140025	181675	**135675**	402	**172920**
B2	16605	20655	**15315**	0	**20640**	X2	6480	7260	6840	0	7260
B3	17460	20565	**16800**	3	**19065**	X3	49650	52680	**45780**	125	**50520**
B4	19035	**26025**	**18975**	1	27330	X4	59560	72860	**59390**	1	**72570**
B5	106290	120840	**104610**	53	**117630**	X5	126465	172500	**124875**	271	**165960**
B6	24450	34215	**24390**	5	**33870**	X6	**6180**	**9480**	6390	452	9600
B7	**28470**	**35640**	28500	0	36660	X7	45000	46680	**44580**	265	**45480**
B8	32820	33030	32820	1	33030	X8	20590	29070	**18890**	5	**27120**
B9	26310	29550	**25695***	40	**27960**	X9	101985	168420	**100920**	16	**159600**
B10	**32790**	**34920**	33765	490	35880	X10	99705	178560	**98970**	342	**165660**

Cette procédure permet d'améliorer 13 valeurs par rapport aux valeurs obtenues avec l'heuristique gloutonne. La valeur encadrée correspond à une amélioration de la meilleure borne supérieure connue. La valeur de borne inférieure avec une étoile signifie que tous les ensembles maximaux ont été énumérés et que cette valeur correspond à l'optimum.

Le calcul de la borne inférieure semble être un bon critère d'évaluation des ensembles d'interventions à sous–traiter car, excepté pour l'instance B4, on observe une corrélation entre la borne inférieure et la borne supérieure. La figure 5.5 expose les différentes valeurs de la borne supérieure en fonction de la borne inférieure dans les résultats obtenus jusqu'à présent.

Fig. 5.5 – Corrélation entre les bornes inférieures et les bornes supérieures trouvées



5.3.3 Enumération des sous–ensembles maximaux par Resolution search

La dernière voie explorée a été l'utilisation de Resolution search pour déterminer l'ensemble d'interventions à sous–traiter. L'expérimentation menée se fonde sur les mêmes principes que la méthode d'énumération présentée précédemment : énumérer des sous–ensembles maximaux et les évaluer en calculant la borne inférieure du sous–problème associé. Notre intuition était que Resolution search permette d'énumérer de manière plus pertinente ces sous–ensembles et ainsi de générer plus rapidement des sous–ensembles donnant une borne inférieure de qualité.

Principe d'exploration

Dans l'implémentation proposée, Resolution search cherche à générer des ensembles maximaux en affectant des valeurs 0 ou 1 à un vecteur u de taille n, n étant le nombre d'interventions.

Si l'intervention I est sous–traitée, $u_I = 1$, si l'intervention I n'est pas sous–traitée, $u_I = 0$ et si le sort de l'intervention I n'est pas décidé, $u_i = *$. Le rôle de la fonction `obstacle` est de trouver un ensemble maximal et d'appeler la fonction `oracle` qui, dans ce cas, consiste à calculer la borne inférieure associée. L'obstacle S représente alors l'ensemble maximal courant et il est ajouté à la famille path–like $\mathscr{F}$ de manière à explorer de nouveaux ensembles dans les itérations futures. De même que pour la méthode d'énumération présentée dans la section précédente, Resolution search explore l'espace des solutions de manière à favoriser l'insertion des interventions isolées et des interventions n'ayant pas de successeurs dans les composantes connexes, tout en prenant en compte un critère d'insertion basé sur un poids attribué à chaque intervention. Comme nous l'avons mentionné dans le chapitre 3, Resolution search prend deux différents critères en considération : (i) un critère d'insertion lors de la phase de descente qui consiste à déterminer quelle intervention sera insérée et (ii) un critère de remise en cause lors du choix du littéral w dans la mise à jour de $\mathscr{F}$ qui consiste à déterminer quelle décision sera remise en cause dans les itérations futures. Nous montrons que l'utilisation de ces deux critères permet d'explorer l'espace des solutions de manière plus adaptée.

Critères d'insertion et de remise en cause

On note $\psi(I)$ le poids d'insertion associé à une intervention I et $\theta(I)$ son poids de remise en cause. Dans les deux cas (insertion ou remise en cause) les interventions sont choisies par ordre décroissant des poids associés. Lors de la création des sous–ensembles, on favorise l'insertion des interventions ayant le moins de successeurs, car la sous–traitance d'une intervention implique la sous–traitance de tous ses successeurs ce qui peut limiter a priori l'insertion d'autres interventions. De plus, on favorise les interventions ayant un ratio $\omega(I)$ important. Le poids d'insertion $\psi(I)$ peut être défini de la manière suivante :

$$\psi(I) = \frac{\omega(I)}{|Succ(I)| + 1}$$

Afin de donner une priorité supplémentaire aux interventions isolées, leur poids a une valeur double :

$$\psi(I) = 2 \cdot \omega(I) \qquad \forall I \text{ tel que } |Pred(I) = 0| \text{ et } |Succ(I) = 0|.$$

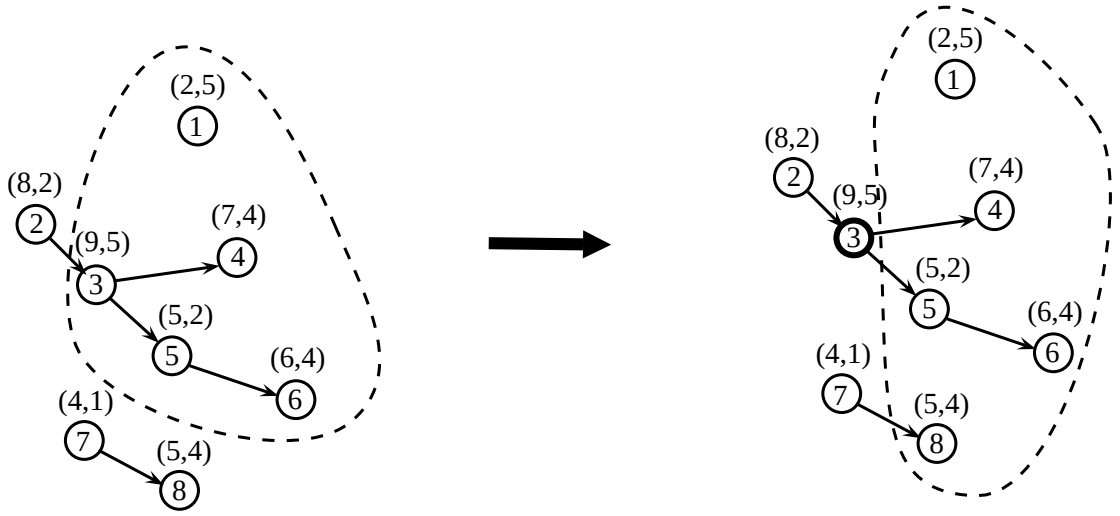
Lorsqu'un ensemble maximal est identifié, il est ajouté à la famille $\mathscr{F}$ sous forme de clause. Cette clause comporte : des instanciations $\bar{x}_I$ signifiant que l'intervention I fait partie de l'ensemble ; des instanciations x_I signifiant que l'intervention I ne doit pas être insérée ; et des variables non instanciées ne faisant pas partie de l'ensemble mais qui pourront être instanciées dans les itérations futures. Durant la mise à jour de $\mathscr{F}$, nous avons vu dans le chapitre 3 qu'il y a deux cas à distinguer : soit $S \not\sqsubseteq \mathscr{F}$ soit $S \sqsubseteq \mathscr{F}$. Dans le premier cas, $R = S$ est simplement ajoutée à $\mathscr{F}$ et dans le deuxième cas, S est simplifiée par résolvantes successives sur certaines clauses de $\mathscr{F}$ pour donner la clause R qui sera insérée dans $\mathscr{F}$.

On distingue deux cas de figure pour la remise en cause d'une instanciation : soit I appartient à l'ensemble et $\bar{x}_I \in R$, soit I n'appartient pas à l'ensemble et $x_I \in R$. Dans les deux cas on choisit un littéral w appartenant à R qui sera fixé à sa valeur opposée dans le prochain vecteur $u(\mathscr{F})$. On préfère choisir un littéral w à valeur 1 dans R ($\bar{x}_I \in R$) pour qu'il soit fixé à 0 dans $u(\mathscr{F})$ de manière à supprimer une intervention de l'ensemble plutôt qu'en ajouter une nouvelle (car l'ensemble est déjà maximal). On préfère également un littéral correspondant à une intervention de faible poids $\psi(I)$ et ayant le moins de prédécesseurs car si l'intervention est retirée de l'ensemble, tous ses prédécesseurs doivent l'être également. Dans le cas où toutes les interventions sont à valeur 0 dans R ($\bar{x}_I \notin R$ pour tout I), le critère de remise en cause est alors directement lié au critère d'insertion, car dans ce cas, choisir w correspond à insérer l'intervention w dans $u(\mathscr{F})$. Si l'on prend en compte les différents critères que l'on vient d'énumérer, le poids de remise en cause $\theta(I)$ lié à une intervention I peut être défini de la manière suivante :

$$\theta(I) = \begin{cases} 1/(\psi(I) + |Pred(I)| + 1) & \text{si } \bar{x}_I \in R \\ -1/\psi(I) & \text{si } x_I \in R \end{cases}$$

Nous allons illustrer la différence de comportement entre l'énumération et Resolution search pour la génération des ensembles maximaux. Comme dans l'exemple précédent, on considère que le budget alloué pour la sous–traitance est $A = 20$. Le schéma de gauche de la figure 5.6 illustre le choix fait par l'énumération pour la création d'un sous–ensemble maximal tel que décrit précédemment.

Fig. 5.6 – Remise en cause du choix des interventions dans le cas de l'énumération



L'intervention 1 étant isolée, elle est choisie en priorité, ensuite les interventions 4 et 6 ayant la plus grande valeur $\omega(I)$ sont sélectionnées et leur prédécesseurs sont insérés jusqu'à ce qu'il ne soit plus possible de rajouter d'interventions. Une fois l'ensemble maximal

$\{1, 3, 4, 5, 6\}$ identifié, la borne inférieure est calculée puis un autre ensemble est généré en supprimant ou en insérant des interventions. Le nouvel ensemble généré par l'énumération est représenté par le schéma de droite. Bien que l'intervention 3 ait un poids $\omega(I)$ important, elle est supprimée de l'ensemble maximal car elle correspond à la dernière intervention insérée précédemment. L'intervention 8, qui semble a priori moins intéressante au regard du critère $\omega(I)$, est insérée pour donner l'ensemble $\{1, 4, 5, 6, 8\}$. On constate ici que le choix de supprimer l'intervention 3 n'est pas très judicieux car, d'une part, cette intervention possède un critère $\omega(I)$ important et, d'autre part, sa suppression empêche l'insertion de son prédécesseur, l'intervention 8, représentant elle aussi un poids $\omega(I)$ important.

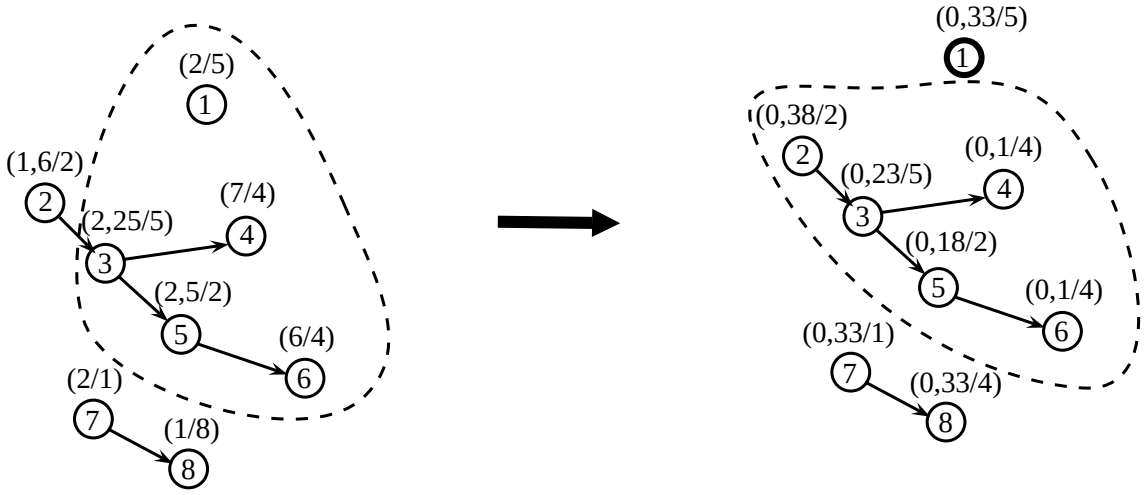
Dans le cas d'une énumération par Resolution search, le calcul des poids $\psi(I)$ et $\theta(I)$ nous donne le tableau 5.4 suivant :

TAB. 5.4 – Poids d'insertion et de remise en cause associés aux interventions

I	1	2	3	4	5	6	7	8
$\psi(I)$	2	1,6	2,25	7	2,5	6	2	1
$\theta(I)$	0,33	0,38	0,23	0,1	0,18	0,1	0,33	0,33

La figure 5.7 illustre le comportement de Resolution search pour la génération des ensembles maximaux.

FIG. 5.7 – Remise en cause du choix des interventions dans le cas de Resolution search



Les valeurs (*val1*,*val2*) correspondent respectivement à $\psi(I)$ et *cost*(I) dans le schéma de gauche et à $\theta(I)$ et *cost*(I) dans le schéma de droite. Le premier ensemble maximal (schéma de gauche) correspond au même ensemble que celui généré par l'énumération.

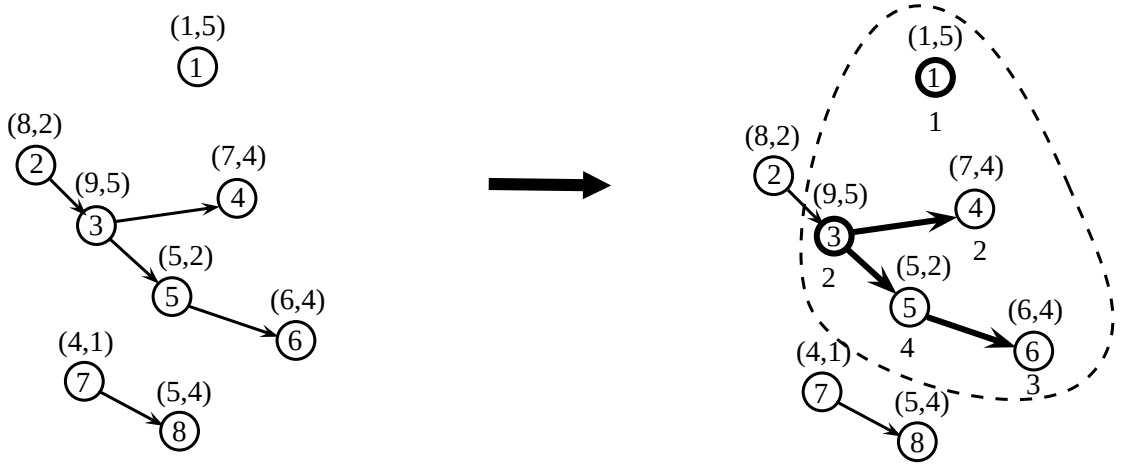
Le deuxième ensemble généré (schéma de droite) est différent : le critère $\theta(I)$ maximal étant celui de l'intervention 1 ($\theta(1) = 0,33$ alors que $\theta(3) = 0,23$), c'est cette intervention qui est supprimée de l'ensemble maximal. L'intervention 2 de valeur $\psi(2) = 1,6$ est alors ajoutée à l'ensemble ce qui nous donne l'ensemble $\{2,3,4,5,6\}$ qui globalement semble meilleur (selon le critère ω) que l'ensemble $\{1,4,5,6,8\}$ généré par l'énumération.

Les critères ψ ou θ n'étant pas optimaux, il se peut que dans certains cas, l'énumération donne de meilleurs sous–ensembles que Resolution search. L'exemple proposé illustre cependant bien le fait que Resolution search permet d'effectuer une recherche plus fine qui ne cantonne pas la remise en cause des choix à l'ordre inverse d'instanciation et peut se baser sur d'autres critères heuristiques.

Phase de remontée implicite

Nous montrons qu'il est possible, dans la résolution de ce problème, d'implémenter la phase de remontée implicite décrite dans le chapitre 4. Reprenons l'exemple précédent en le modifiant légèrement : supposons que le poids $\psi(3)$ soit égal à 8 au lieu de $2,25$. La première intervention insérée est donc l'intervention 3. On peut déduire alors par implication que les interventions 4,5 et 6 doivent également être insérées. Comme nous l'avons démontré dans la proposition 4.1 du chapitre 4, puisque $\bar{x}_3 \Rightarrow \bar{x}_4\bar{x}_5\bar{x}_6$ alors $S = \bar{x}_3$. L'intervention qui suit est l'intervention 1 de poids $\psi(1) = 2$ et l'ensemble {1,3,4,5,6} résultant est maximal. L'ensemble maximal généré est alors représenté par la clause $S = \bar{x}_1\bar{x}_3$ et la remise en cause ne porte que sur x_1 ou x_3. Il est en effet inutile d'essayer de supprimer les interventions 4, 5 ou 6 tant que l'intervention 3 est dans l'ensemble. L'application de la phase de remontée implicite permet ici d'identifier plus rapidement quels choix méritent d'être remis en cause ou non. La figure 5.8 illustre le fonctionnement de la phase de remontée implicite dans la construction d'un ensemble maximal.

Fig. 5.8 – Illustration de la phase de remontée implicite



Résultats expérimentaux

Les résultats expérimentaux obtenus en exécutant Resolution search durant 600 secondes sont exposés dans le tableau 5.5. Ces résultats montrent que Resolution search se comporte mieux de manière générale que l'énumération de type backtracking pour trouver un sous–ensemble générant une borne inférieure de qualité. Dans ce tableau, `BI` correspond à la meilleure borne inférieure trouvée, t(s) correspond au temps cpu en secondes mis pour trouver la borne et `BS` correspond à la valeur de l'objectif trouvé par l'algorithme GRASP après 1200 secondes. La colonne *meilleures valeurs* correspond aux meilleures valeurs obtenues jusqu'à présent avec l'heuristique gloutonne ou l'énumération et *Resolution search* correspond aux valeurs trouvées par Resolution search.

TAB. 5.5 – Résultats obtenus par Resolution search

Pb	meilleures valeurs		Resolution search			Pb	meilleures valeurs		Resolution search		
	BI	BS	BI	t(s)	BS		BI	BS	BI	t(s)	BS
B1	37845	43860	**36525**	172	**42180**	X1	**135675**	**172920**	**135675**	401	**172920**
B2	15315	20640	**15225**	0	**20340**	X2	**6480**	**7260**	**6840**	0	**7260**
B3	**16800**	**19065**	16965	0	19410	X3	45780	50520	**45360**	308	**50280**
B4	18975	**26025**	**18915**	1	26265	X4	59390	72570	**59380**	64	**71560**
B5	**104610**	**117630**	106230	113	120030	X5	**124875**	**165960**	**124875**	42	**165960**
B6	**24390**	33870	**24390**	0	**33390**	X6	**6180**	9480	**6180**	36	**9390**
B7	28470	35640	**27930**	286	**35280**	X7	44580	45480	**43320**	460	**43680**
B8	**32820**	**33030**	**32820**	0	35760	X8	18890	27120	**18830**	413	**25680**
B9	**25695***	**27960**	**25695***	5	**27960**	X9	**100920**	**159600**	**100920**	16	164400
B10	**32790**	**34920**	**32790**	0	35040	X10	98970	165660	**98010**	525	**160920**

Sur les 20 instances testées, Resolution search améliore strictement 9 bornes inférieures par rapport à l'heuristique gloutonne et l'énumération et trouve au total 18 meilleures bornes inférieures. Les valeurs encadrées correspondent à une amélioration des meilleures valeurs connues et la valeur avec une étoile correspond à l'optimum. On remarque que dans le cas d'une égalité avec l'énumération, Resolution search est toujours plus rapide pour trouver le meilleur ensemble maximal. Dans certains cas, malgré une borne inférieure plus basse ou égale, les valeurs de la fonction objective sont moins bonnes. Ceci est dû aux raisons suivantes : (i) l'évaluation d'un sous–ensemble par le calcul de la borne inférieure n'est pas optimal, il se peut donc qu'un ensemble donnant une borne inférieure plus basse donne une valeur optimale de moins bonne qualité (cf. instances B4 et B5) et (ii) dans certains cas, deux sous–ensembles différents donnent des bornes de même valeur et donc peuvent amener à une valeur de l'objectif différente (cf. instances X9, B6 et B10).

Utilisation d'une relaxation du problème

Une autre méthode expérimentée consiste à utiliser une relaxation du problème pour identifier les poids d'insertion $\psi(I)$. Considérons les variables de décision $x_{I,t} = 1$ si l'intervention I est affectée au technicien t, et $y_I = 1$ si l'intervention I est sous–traitée. On propose de formuler la relaxation $\tilde{P}$ du problème P de la manière suivante :

$$
\begin{aligned}
(\tilde{P}) \text{ Minimiser} \quad & \rho \\
\text{sujet à} \quad & \sum \{x_{I,t} \mid C(t,i) \geq n\} \geq (1 - y_I) R(I,i,n), \quad \forall I, \forall i, \forall n, && (5.15) \\
& \sum_I T(I) x_{I,t} \leq \rho \cdot H_{max}, \quad \forall t, && (5.16) \\
& x_{I,t} + y_I \leq 1, \quad \forall I, \forall t, && (5.17) \\
& y_I \leq y_J, \quad \forall I \in Pred(J), && (5.18) \\
& \sum_I cost(I) y_I \leq A, && (5.19) \\
& x_{I,t} \geq 0, \quad \forall I, \forall t, && (5.20) \\
& y_I \geq 0, \quad \forall I. && (5.21)
\end{aligned}
$$

Les contraintes (5.15) vérifient le respect des compétences requises par l'intervention ; les contraintes (5.16) assurent que chaque technicien ne travaille pas plus que le temps global de l'ordonnancement ; les contraintes (5.17) imposent qu'une intervention est : soit sous–traitée, soit traitée par au moins un technicien ; les contraintes (5.18) maintiennent les relations de précédence entre les interventions sous–traitées et la contrainte (5.19) veille au respect de la contrainte de budget pour sous–traiter les interventions.

Le problème $\tilde{P}$ est une relaxation du problème P original car les journées et les équipes de techniciens ne sont plus prises en compte. La résolution de $\tilde{P}$ nous donne un ordonnancement qui minimise le nombre de jours en prenant simplement en compte l'affectation des techniciens aux interventions tout en veillant au respect des contraintes de précédence et au respect de la contrainte de budget. Nous proposons d'utiliser $\tilde{P}$ pour l'initialisation du critère d'insertion $\psi(I)$. Pour cela, nous considérons de manière empirique qu'il est préférable de sous–traiter une intervention I de valeur fractionnaire $y_I > 0$ à l'optimum de $\tilde{P}$ plutôt qu'une intervention à valeur $y_I = 0$. Notre expérimentation a consisté à résoudre $\tilde{P}$ et à initialiser les poids d'insertion $\psi(I)$ de la manière suivante :

$$
\psi(I) = \begin{cases} \mathbf{M} + \omega(I)/(|Succ(I)| + 1) & \text{si } y_I > 0 \\ \omega(I)/(|Succ(I)| + 1) & \text{sinon} \end{cases}
$$

avec $\mathbf{M}$ la valeur maximale des ratios $\omega(I)/(|Succ(I)|+1)$ parmi toutes les interventions I. Ainsi, les interventions de valeur $y_I > 0$ sont insérées en priorité et seront classées entre elles selon l'ordre décroissant des valeurs $\psi(I)$ et les autres interventions sont classées en dernier suivant l'ordre décroissant des valeurs $\psi(I)$. Les résultats obtenus en exécutant cette version de l'algorithme pendant 600 secondes sont exposés dans le tableau 5.6. Comme dans les tableaux de résultats précédemment exposées, les valeurs `BS` correspondent à la meilleure valeur de la fonction objective trouvée en exécutant l'algorithme GRASP pendant 1200 secondes.

TAB. 5.6 – Résultats obtenus par Resolution search avec prise en compte de $\tilde{P}$

Pb	Meilleures valeurs		Resolution search+$\tilde{P}$			Pb	Meilleures valeurs		Resolution search+$\tilde{P}$		
	BI	BS	BI	t(s)	BS		BI	BS	BI	t(s)	BS
B1	**36525**	**42180**	38565	120	44430	X1	**135675**	**172920**	**135675**	93	**172920**
B2	**15225**	**20340**	19915	0	**20340**	X2	6480	7260	**6420**	0	**6840**
B3	**16800**	**19065**	17715	0	27015	X3	45360	50280	**44250**	0	**50160**
B4	**18915**	**26025**	**18915**	1	26265	X4	59380	**71560**	**58740**	416	71800
B5	104610	117630	**93360**	0	**104760**	X5	124875	165960	**124875**	282	**165960**
B6	**24390**	**33390**	24870	0	34920	X6	**6180**	**9390**	6825	46	9465
B7	**27930**	**35280**	28470	311	36600	X7	**43320**	**43680**	44340	360	45240
B8	**32820**	33030	**32820**	0	**33000**	X8	**18830**	**25680**	19210	100	27300
B9	**25695**	**27960**	**25695**	2	**27960**	X9	100920	**159600**	**100545**	23	161640
B10	**32790**	**34920**	**32790**	0	35040	X10	98010	**160920**	**92355**	190	172800

Dans l'ensemble, les résultats ne sont pas meilleurs que ceux donnés par la première implémentation de Resolution search, cependant la prise en compte de la solution optimale de $\tilde{P}$ nous donne certaines améliorations importantes. Tout d'abord, on constate une nette diminution de la borne inférieure trouvée pour l'instance B5 qui passe de 104610 à 93360. De plus, on obtient des améliorations des meilleures solutions connues pour les instances B8 et X2. On remarque encore une fois que la corrélation entre la valeur de la borne inférieure et celle de la borne supérieure n'est pas toujours vérifiée. On observe, par exemple, une amélioration de la borne supérieure pour l'instance B8 alors que la valeur de la borne inférieure est la même que pour la version précédente de Resolution Search (tableau 5.5).

Preuves d'optimalité

Nous avons mené une dernière expérimentation consistant à exécuter l'algorithme de recherche des ensembles maximaux par Resolution search durant 10 heures. Les objectifs étaient de savoir jusqu'à quel niveau cette préprocédure pouvait améliorer les résultats obtenus par la méthode GRASP et s'il était possible d'énumérer tous les ensembles maximaux pour d'autres instances que B9. Nous avons exécuté les deux versions étudiées précédemment : la version classique et la version avec prise en compte de la solution optimale de $\tilde{P}$. Les résultats sont exposés dans le tableau 5.7. La colonne *Meilleures valeurs* correspond aux meilleures valeurs obtenues jusqu'à présent, la colonne *Resolution search* correspond aux valeurs obtenus par Resolution search et la colonne *Resolution search* + $\tilde{P}$ aux valeurs obtenues par Resolution search avec prise en compte de la solution optimale de $\tilde{P}$ (les deux variantes ayant été exécutées pendant 10 heures). La colonne *challenge* correspond aux meilleures valeurs obtenues parmi tous les participants au challenge. Les colonnes `BI` et `BS` correspondent respectivement aux bornes inférieures et supérieures obtenues et la colonne *t(s)* est le temps d'exécution en secondes nécessaire à l'obtention de la borne inférieure.

Les valeurs en gras indiquent une amélioration de la borne supérieure par rapport aux meilleures valeurs obtenues jusqu'à présent et les valeurs encadrées correspondent aux améliorations par rapport aux meilleurs résultats obtenus parmi les participants au challenge. Les valeurs de bornes inférieures ayant une étoile correspondent aux valeurs optimales.

Tab. 5.7 – Résultats obtenus par Resolution search après 10h de temps de calculs

	Meilleures valeurs		Resolution search			Resolution search + $\tilde{P}$			challenge
	BI	BS	BI	t (s)	BS	BI	t (s)	BS	BS
B1	36525	42180	36240	1066	**41730**	37455	5441	42630	34395
B2	15225	20340	15015	605	18915	15135	21458	**18705**	15870
B3	16800	**19065**	16770	5403	20565	17715	0	21375	16020
B4	18915	26025	18495	4847	26325	18195	4631	**25965**	25305
B5	93360	**104760**	104310	13654	115260	93360	0	**104760**	89700
B6	24390	**33390**	24390	0	**33390**	24855	2304	34545	27615
B7	27930	**35280**	27930	286	35760	28050	10899	35940	33300
B8	32820	**33000**	32820	0	33030	32820	0	**33000**	33030
B9	25695	**27960**	25695*	4	**27960**	25695*	2	**27690**	28200
B10	32790	**34920**	32790	0	35040	32790	0	35040	34680
X1	135675	**172920**	135675*	510	**172920**	135675*	93	**172920**	151140
X2	6480	**6840**	6420	3435	7050	6270	864	6990	7260
X3	45360	50280	43830	1404	**50160**	43980	1404	**50160**	50040
X4	59380	**71560**	59060	12764	71640	58740	416	71800	65400
X5	124875	**165960**	124875*	282	**165960**	124875*	282	**165960**	147000
X6	6180	**9390**	6180	36	**9390**	6825	46	9465	9480
X7	43320	43680	42990	26364	43560	41310	31091	**41880**	33240
X8	18830	**25680**	18190*	20571	25740	18190*	20205	25800	23640
X9	100920	159600	99015*	1279	**153660**	99015*	1633	167880	134760
X10	98010	**160920**	96750	27011	175440	91500	33110	168060	137040

On constate une nette amélioration des valeurs de borne inférieure (11 valeurs sur les 20 instances ont été améliorées). Cependant les bornes supérieures ne suivent pas toujours cette tendance car seules les instances B1, B2, B4, X3, X7 et X9 sont strictement améliorées. On constate de nouveau que l'évaluation des sous–ensembles maximaux par le calcul de la borne inférieure n'est pas toujours fiable : pour les instances B3, X2, X4, X8 et X10 une borne inférieure plus petite donne une borne supérieure plus grande.

5.4 Conclusion

Dans ce chapitre, nous avons présenté un problème de planification de techniciens et d'interventions pour les télécommunications. Nous avons proposé une heuristique de résolution constituée de trois phases : (i) la fixation de certaines variables par la résolution heuristique d'un problème de sac à dos avec contraintes de précédence pour le choix des interventions à sous–traiter, (ii) l'initialisation de la mémoire (poids attribués aux interventions) réalisée par la recherche de la meilleure permutation des poids associés aux priorités des interventions et (iii) la procédure GRASP qui cherche à améliorer les solutions initiales tout en mettant à jour les poids attribués aux interventions.

Les expérimentations ont montré que le choix de l'ensemble d'interventions à sous–traiter peut directement influencer la qualité des résultats produits par notre approche. Nous avons donc proposé une étude portant sur différentes manières d'identifier ce sous–ensemble et avons en particulier montré que Resolution search offre d'intéressantes possibilités d'exploration. Contrairement à un algorithme d'énumération classique (de type backtracking), Resolution search permet d'énumérer les ensembles maximaux suivant deux critères : un critère d'insertion et un critère de remise en cause. Nous avons proposé différentes manières d'implémenter ces critères en fonction du nombre minimum de techniciens requis et du nombre de prédécesseurs ou de successeurs des interventions. Les résultats expérimentaux montrent clairement l'intérêt de la prise en compte de ces deux critères pour la résolution de ce sous–problème. De manière plus générale, cela montre également les possibilités intéressantes qu'offre Resolution search pour la résolution de problèmes d'optimisation combinatoire. Une perspective intéressante serait d'étudier de manière plus approfondie l'affectation des critères d'insertion et de remise en cause pour une énumération plus pertinente de l'espace des solutions. Il serait également intéressant de trouver d'autres manières d'évaluer la qualité des sous–ensembles maximaux car l'évaluation de ces sous–ensembles par le calcul de la borne inférieure a montré, dans certains cas, qu'il était sous–optimal.

Chapitre 6

Conclusion et perspectives

Les problèmes d'optimisation linéaire à variables binaires permettent de modéliser un grand nombre de problèmes industriels et leur résolution constitue un enjeu important. Notre thèse contribue à améliorer la résolution exacte de ces problèmes en s'appuyant sur l'application de Resolution search à la résolution du problème du sac à dos multidimensionnel en 0–1 (MKP).

Dans un premier temps, nous avons proposé un algorithme d'énumération implicite s'appuyant sur le principe du backtracking pour le MKP. Cet algorithme intègre deux contraintes additionnelles au modèle initial : la première est générée à partir des coûts réduits, des valeurs optimales et d'un minorant quelconque du problème ; la deuxième fixe le nombre d'objets à une valeur entière. Nous avons montré que l'utilisation simultanée de ces deux contraintes permet de fixer davantage de variables à leur valeur optimale et d'améliorer l'élagage de l'arbre de recherche. Nous avons également vu que la contrainte sur le nombre d'objets permet de générer des contraintes utiles pour résoudre des sous–problèmes ayant peu de variables par une énumération efficace. La stratégie de branchement originale que nous proposons vise à explorer en priorité les affectations partielles susceptibles de violer la contrainte sur les coûts réduits afin d'élaguer au plus tôt l'arbre de recherche. Cet algorithme résout des instances difficiles du jeu Chu et Beasley de la `OR-Library` à 5 contraintes / 500 variables en un temps plus rapide que les algorithmes de l'état de l'art connus et résout l'ensemble des instances à 10 contraintes / 250 variables pour lesquelles la plupart des valeurs optimales étaient inconnues jusqu'alors. Nous avons montré cependant que son efficacité est conditionnée par la connaissance d'un bon minorant et qu'elle est inopérante, même pour des petites instances (5 contraintes / 100 variables), si aucun minorant n'est connu.

Dans un deuxième temps, nous avons étudié Resolution search, une méthode de résolution centrée sur un schéma d'exploration plus original inspiré des backtrackings intelligents. Cette méthode offre des possibilités d'exploration intéressantes : non seulement elle mémorise les échecs rencontrés de manière à limiter le *trashing* mais elle exploite également ces échecs afin de guider la recherche vers des affectations partielles prometteuses.

Elle offre, de plus, la possibilité d'implémenter *deux* stratégies de branchement distinctes : (i) *une stratégie d'affectation* qui consiste à déterminer quelle variable sera instanciée et à quelle valeur elle le sera lors de la recherche de solutions et (ii) *une stratégie de remise en cause* qui consiste à déterminer quelle instanciation sera remise en cause dans les itérations futures. Cette caractéristique est intéressante car, contrairement à la plupart des algorithmes d'énumération, Resolution search ne limite pas la remise en cause des instanciations passées selon l'ordre chronologique mais permet de prendre en compte un critère heuristique plus adapté au problème traité. Expérimentalement, elle obtient rapidement de bons minorants (d'une manière comparable aux meilleures heuristiques connues). Cependant, l'énumération implicite proposée précédemment résout plus rapidement l'ensemble des instances testées si un bon minorant de départ lui est fourni.

Ces différentes observations nous ont amené, dans un troisième temps, à concevoir un algorithme combinant Resolution search avec une énumération implicite inspirée de celle présentée précédemment. Nous avons tout d'abord proposé une alternative au processus d'identification des obstacles minimaux qui, contrairement à la méthode originale, procède en temps linéaire sans faire de multiples appels successifs à l'algorithme du simplexe. Nous avons également implémenté une version itérative de Resolution search qui diversifie la recherche en explorant partiellement, et de manière répétée, plusieurs sous–problèmes du problème initial. Cet algorithme coopératif utilise Resolution search pour guider l'exploration vers des zones prometteuses de l'espace de recherche et l'énumération implicite pour résoudre de manière efficace des sous–problèmes de petite taille. Ce schéma d'exploration présente une réelle coopération entre les deux méthodes ; Resolution search fournit des sous–problèmes à résoudre à l'énumération implicite qui, à son tour, favorise l'obtention d'obstacles minimaux pour guider la recherche. Les résultats obtenus sur les instances à 5-10 contraintes / 250 variables et 5 contraintes / 500 variables du jeu Chu et Beasley de la `OR-Library` ont démontré les qualités de cette approche qui, non seulement, trouve de bons minorants d'une manière comparable aux meilleures heuristiques de la littérature, mais résout également les instances à l'optimum en un temps plus rapide que les meilleures méthodes exactes connues. En outre, elle obtient les preuves d'optimalité des 30 instances à 10 contraintes / 500 variables. Ces valeurs optimales étaient inconnues jusqu'à présent.

Finalement, Resolution search a été appliqué à un problème de planification de techniciens et d'interventions pour les télécommunications. Nous avons tout d'abord présenté un algorithme de résolution de ce problème fondé sur la métaheuristique GRASP. Nous nous sommes ensuite penchés sur l'application de Resolution search à un sous–problème consistant à énumérer et évaluer des sous–ensembles d'interventions. Nous avons proposé des stratégies d'affectation et de remise en cause spécifiques qui prennent en compte différents critères liés aux interventions. Cela nous a permis d'obtenir de meilleurs résultats que ceux donnés par une méthode d'énumération plus classique de type backtracking. Cette application montre les possibilités intéressantes qu'offre Resolution search pour la résolution de problèmes d'optimisation *combinatoire* à variables binaires au delà du cadre des

problèmes d'optimisation *linéaire* à variables binaires.

Plusieurs voies de recherche seraient intéressantes à poursuivre par la suite. En ce qui concerne la résolution du MKP par notre approche hybride, nous envisageons d'étudier les possibilités d'échange d'information entre chaque sous–problème associé à un nombre d'objets fixé. Dans la version actuelle, ces sous–problèmes sont résolus de manière indépendante et la seule information partagée est le meilleur minorant connu. Il serait intéressant d'étudier par exemple les possibilités d'échange d'obstacles entre les familles path-like associées à chaque sous–problème. Nous envisageons également d'étudier d'autres structures de mémorisation que la famille path–like. Sa complexité spatiale en $O(n)$ est obtenue au prix de « l'oubli » d'une grande partie de l'information acquise pendant la recherche. Une famille contenant plus d'obstacles permettrait de conserver davantage d'information. Enfin, il serait intéressant de poursuivre les travaux initiés par Palpant et al. [53] sur l'application de Resolution search aux problèmes de satisfaction de contraintes. On pourrait imaginer par exemple un schéma de coopération *Resolution search / DPLL* (algorithme de backtracking introduit par Davis, Putnam, Logemann et Loveland [18]) inspiré du schéma *Resolution search / énumération implicite* présenté dans ce mémoire.

Liste des publications

Revues internationales

- S. Boussier, H. Hashimoto, M. Vasquez et C. Wilbaut. « Un algorithme GRASP pour le problème de planification de techniciens et d'interventions pour les télécommunications ». à paraître dans *RAIRO Operations Research* (2008).

- Y. Vimont, S. Boussier et M. Vasquez. « Reduced costs propagation in an efficient implicit enumeration for the 0–1 multidimensional knapsack problem ». *Journal Of Combinatorial Optimization* 15(2) : 165-178 (2008).

Soumissions à des revues internationales

- S. Boussier, M. Vasquez, Y. Vimont, S. Hanafi et P. Michelon. « Combining Resolution Search and Branch & Bound for the 0-1 Multidimensional Knapsack Problem ». Soumis à *Discrete Applied Mathematics* (2008).

- H. Hashimoto, S. Boussier, M. Vasquez et C. Wilbaut. « A GRASP-Based Approach for Technicians and Interventions Scheduling for Telecommunications ». soumis à *Annals of Operations Research* (2008).

Communications internationales

- S. Boussier, H. Hashimoto, M. Vasquez et C. Wilbaut « Some improvements for Technicians and Interventions Scheduling for Telecommunications ». *Workshop on Metaheuristics for Logistics and Vehicle Routing* (EU/MEeting 2008), Troyes, Octobre 2008.

- S. Boussier, M. Vasquez et Y. Vimont. « Application de resolution search au problème du sac à dos multidimensionnel en 0-1 ». *7 ème Conférence Internationale de Modélisation et Simulation* (MOSIM'08). Paris, mars 2008.

- S. Boussier, H. Hashimoto et M. Vasquez. « A Greedy Randomized Adaptive Search Procedure for Technicians and Interventions Scheduling for Telecommunications ». *Metaheuristics International Conference* (MIC2007). Montréal, juin 2007.

- S. Boussier, Y. Vimont et M. Vasquez. « Exploitation de la contrainte de coûts réduits pour la résolution exacte du sac à dos multidimensionnel en 0-1 ». *Conférence scientifique conjointe en Recherche Opérationnelle et Aide à la Décision* (FRANCORO V / ROADEF'07). Grenoble, février 2007.

Soumission à une conférence internationale

- S. Boussier, Y. Vimont, M. Vasquez, S. Hanafi et P. Michelon. « Solving the 0–1 Multidimensional Knapsack Problem with Resolution Search ». *VI ALIO/EURO Workshop on Applied Combinatorial Optimization*. Buenos Aires, Décembre 2008.

Communications nationales

- S. Boussier, M. Vasquez et Y. Vimont. « Resolution Search avec prise en compte des coûts réduits pour le problème du sac à dos multidimensionnel en 0-1 ». *9 ème congrès de la Société Française de Recherche Opérationnelle et d'Aide à la Décision* (ROADEF'08). Clermont-Ferrand, février 2008.

- S. Boussier, Y. Vimont et M. Vasquez. « Propagation des coûts réduits et énumération implicite pour le problème du sac à dos multidimensionnel en 0-1 ». *2 èmes Journées Francophones de Programmation par Contraintes* (JFPC'06). Nîmes, juin 2006.

Liste des figures

Liste des tableaux

Liste des algorithmes

Bibliographie

[1] J.B. Atkinson. « A greedy randomised search heuristic for time-constrained vehicle scheduling and the incorporation of a learning strategy ». *Journal of the Operational Research Society*, 49 :700–708, 1998.

[2] A. B. Baker. « *Intelligent backtracking on constraint satisfaction problems : experimental and theoretical results* ». Thèse de Doctorat, University of Oregon, 1995.

[3] E. Balas et C.H. Martin. « Pivot and complement a heuristic for zero-one programming ». *Management Science*, 26[1] :86–96, 1980.

[4] J.E. Beasley. « OR-Library : Distributing test problems by electronic mail. ». *J. Operational Research Society*, 41 :1069–1072, 1990.

[5] R. Bellman et D. Stuart. *Applied Dynamic Programming.* Princeton University Press, 1962.

[6] D. Bertsimas et R. Demir. « An Approximate Dynamic Programming Approach to Multidimensional Knapsack Problems ». *Management Science*, 48 :550–565, 2002.

[7] J.R. Bitner et E. Reingold. « Backtrack Programming Techniques ». *Communications of the ACM*, 18[11] :651–656, 1975.

[8] S. Boussier, H. Hashimoto, M. Vasquez et C. Wilbaut. « un algorithme GRASP pour le problème de planification de techniciens et d'interventions pour les télécommunications ». *soumis à RAIRO Operations Research*, 2008.

[9] S. Boussier, M. Vasquez, Y. Vimont, S. Hanafi et P. Michelon. « Combining Resolution Search and Branch & Bound for the 0-1 Multidimensional Knapsack Problem ». *soumis à Discrete Applied Mathematics*, 2008.

[10] M. Bruynooghe et R. Venken. « Backtracking ». *Encyclopedia of Artificial Intelligence (2nd Edition), Wiley, New York*, pages 84–88, 1992.

[11] P.C. Chu et J.E Beasley. « A genetic algorithm for the multidimensional knapsack problem. ». *Journal of Heuristics*, [4] :63–86, 1998.

[12] V. Chvátal. *Linear Programming.* W.H. Freeman and Company, 1983.

[13] V. CHVÁTAL. « Resolution search ». Rapport Technique, DIMACS 95-14, 1995.

[14] V. CHVÁTAL. « Resolution search ». *Discrete Applied Mathematics*, 73 :81–99, 1997.

[15] G. CODATO et M. FISCHETTI. « Combinatorial Benders' Cuts for Mixed-Integer Linear Programming ». *Operations Research*, 54 :756–766, 2006.

[16] T.H. CORMEN, C.E. LEISERSON, R.L. RIVEST et C. STEIN. *Introduction to algorithms.* The MIT Press, 2 édition, 2001.

[17] G.G. DANTZIG. « Discrete-variable extremum problems ». *Operations Research*, 5 :266–277, 1957.

[18] M. DAVIS, G. LOGEMANN et D. LOVELAND. « A machine program for theorem-proving ». *Communications of the ACM*, 5 :394–397, 1962.

[19] R. DECHTER. « Enhancement schemes for constraint processing : Backjumping, learning and cutset decomposition ». *Artificial Intelligence*, 41 :273–312, 1990.

[20] S. DEMASSEY. « *Méthodes hybrides de programmation par contraintes et programmation linéaire pour le Problème d'ordonnancement de Projet à contraintes de ressources* ». Thèse de Doctorat, Université d'Avignon, 2003.

[21] S. DEMASSEY, C. ARTIGUES et P. MICHELON. « An application of resolution search to the rcpsp ». Dans *17th European Conference on Combinatorial Optimization ECCO'04*, Beyrouth, Lebanon, 2004.

[22] D. FAYARD et G. PLATEAU. « An algorithm for the solution of the 0-1 Knapsack Problem ». *Computing*, 28[3] :269–287, 1981.

[23] C. FLEURENT et F. GLOVER. « Improved constructive multistart strategies for the quadratic assignment problem using adaptive memory ». *INFORMS Journal on Computing*, 11 :198–204, 1999.

[24] A. FREVILLE et S. HANAFI. « The Multidimensional 0-1 Knapsack Problem - Bounds and Computational Aspects ». *Annals of Operations Research*, 139 :195–227, 2005.

[25] A. FREVILLE et G. PLATEAU. « Hard 0-1 multiknapsack test problems for size reduction methods ». *Investigation Operativa*, 1 :251–270, 1990.

[26] A. FRÉVILLE. « *Contribution à l'optimisation en nombre entiers* ». Thèse de Doctorat, Université Paris XIII, 1991.

[27] A. FRÉVILLE. « The Multidimensional 0-1 knapsack problem : An overview ». *European Journal of Operational Research*, 155 :1–21, 2004.

[28] A. Fréville et G. Plateau. « Sac à Dos Multidimensionnel en variables 0-1 : Encadrement de la Somme des Variables à l'optimum ». *RAIRO. Recherche Opérationnelle*, 27 :169–187, 1993.

[29] A. Fréville et G. Plateau. « An efficient preprocessing procedure for the multidimensional 0-1 knapsack problem ». *Discrete Applied Mathematics*, 49 :189–212, 1994.

[30] J. Gaschnig. « Experimental case studies of backtrack vs. Waltz-type vs.new algorithms for satisficing assignement problems ». *Proceedings of teh Second Conference of the Canadian Society for Cumputational Studies of Intelligence, Toronto*, 1978.

[31] J. Gaschnig. « *Performance Measurement and Analysis of Certain Search Algorithms* ». Thèse de Doctorat, Carnegie-Mellon University, Pittsburgh, PA,, 1979.

[32] B. Gavish et H. Pirkul. « Allocation of data bases and processors in a distributed computing system ». *Management od Distributed Data Processing*, 31 :215–231, 1982.

[33] B. Gavish et H. Pirkul. « Efficient algorithms for solving multiconstraint zero-one knapsack problems to optimality ». *Mathematical Programming*, 31 :78–105, 1985.

[34] P.C. Gilmore et R.E. Gomory. « The theory and computation of knapsack functions ». *Operations Research*, 14 :1045–1074, 1966.

[35] M.L. Ginsberg. « Dynamic Backtracking ». *Journal of Artificial Intelligence Research*, 1 :25–46, 1993.

[36] F. Glover. « A Multiphase-Dual Algorithm forthe Zero-One Integer Programming Problem ». *Operations Research*, 13[6] :879–919, 1965.

[37] F. Glover et L. Tangedahl. « Dynamic Strategies for Branch and Bound ». *OMEGA, The Int. J. of Mgmt Sci*, 4[5] :571–575, 1976.

[38] M. Gondran et M. Minoux. *Graphes & algorithmes*. Eyrolles, 1985.

[39] S. Hanafi et A. Freville. « An efficient tabu search approach for the 0-1 multidimensional knapsack problem ». *European Journal Of Operational Research*, 106 :659–675, 1998.

[40] S. Hanafi et F. Glover. « Resolution Search and Dynamic Branch-And-Bound ». *Journal of Combinatorial Optimization*, 6 :401–423, 2002.

[41] S. Hanafi et C. Wilbaut. « Scatter search for the 0-1 multidimensional knapsack problem ». *Journal of Mathematical Modelling and Algorithms*, 7[2] :143–159, 2008.

[42] H. HASHIMOTO, S. BOUSSIER, M. VASQUEZ et C. WILBAUT. « A GRASP-Based Approach for Technicians and Interventions Scheduling for Telecommunications ». *soumis à Annals Of Operations Research*, 2008.

[43] R.J.W. JAMES et Y. NAKAGAWA. « Enumeration Methods for Repeatedly Solving Multidimensional Knapsack Sub-Problems ». Rapport Technique 10, The Institute of Electronics, Information and Communication Engineers, october 2005.

[44] H. KELLERER, U. PFERSCHY et D. PISINGER. *Knapsack Problems*. Springer, 2004.

[45] R. E. KORF. Search Techniques. Dans *Encyclopedia of Information Systems*, pages 31–43. Elsevier, New York, 2002.

[46] A. H. LAND et A. DOIG. « An automatic method of solving discrete programming problems ». *Econometrica*, 28 :497–520, 1960.

[47] A. LODI, S. MARTELLO et D. VIGO. « Models and Bounds for Two-Dimensional Level Packing Problems ». *Journal of Combinatorial Optimization*, 8 :363–379, 2004.

[48] J. LORIE et L.J. SAVAGE. « Three problems in capital rationing ». *Journal of Business*, 28[3] :229–239, 1955.

[49] A. S. MANNE et H. M. MARKOWITZ. « On the solution of discrete programming problems ». *Econometrica*, 25 :84–110, 1957.

[50] 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.

[51] C. OLIVA, P. MICHELON et C. ARTIGUES. « Constraint and Linear Programming : Using Reduced Costs for solving the Zero/One Multiple Knapsack Problem ». *CP 01, Proceedings of the workshop on Cooperative Solvers in Constraint Programming(CoSolv 01)*, pages 87,98, 2001.

[52] M.A. OSORIO, F. GLOVER et P. HAMMER. « Cutting and surrogate constraint analysis for improved multidimensional knapsack solutions ». *Annals Of Operations Research*, 117 :71–93, 2002.

[53] M. PALPANT. « *Recherche exacte et approchée en optimisation combinatoire : schémas d'intégration et applications.* ». Thèse de Doctorat, Université d'Avignon, 2005.

[54] M. PALPANT, C. ARTIGUES et C. OLIVA. « MARS : a hybrid scheme based on resolution search and constraint programming for constraint satisfaction problems, LAAS report 07194 ». Rapport Technique, LAAS-CNRS, Université de Toulouse, France, 2007.

[55] C. C. PETERSEN. « Computational experience with variants of the Balas algorithm applied to the selection of R&D projects ». *Management Science*, 13 :736–750, 1967.

[56] D. Pisinger. « *Algorithms for Knapsack Problems* ». Thèse de Doctorat, Dept of Computer Science University of Copenhagen, 1995.

[57] L.S. Pitsoulis et M.G. Resende. « Greedy Randomized Adaptive Search Procedures ». Rapport Technique, AT&T Labs Research, 2001.

[58] J. Puchinger, G.R. Raidl et U. Pferschy. « *Evolutionary Computation in Combinatorial Optimization* », Chapitre The Core Concept for the Multidimensional Knapsack Problem, pages 195–208. Springer, 2006.

[59] M.G.C. Resende et C.C. Ribeiro. « A GRASP for graph planarization ». *Networks*, 29 :173–189, 1997.

[60] J. A. Robinson. « A machine-oriented logic based on the resolution principle ». *J. ACM*, 12 :23–41, 1965.

[61] R. M. Saunders et R. Schinzinger. « A Shrinking Boundary Algorithm for Discrete System Models ». *IEEE Transactions On Systems Science And Cybernetics*, ssc-6[2] :133–140, 1970.

[62] S. Senju et Y. Toyoda. « An approach to linear programming with 0-1 variables ». *Management Science*, 15 :196–207, 1968.

[63] W. Shih. « A branch and bound method for the multiconstraint zer-one knapsack problem ». *Journal of Operational Reasearch Society*, 30 :369–378, 1979.

[64] R.M. Stallman et G.J. Sussman. « Forward reasoning and dependency-directed-backtracking in a system for computer-aided circuit analysis ». *Artificial Intelligence*, 9 :135–196, 1977.

[65] E.D. Taillard, L.M. Gambardella, M. Gendreau et J.Potvin. « Adaptive memory programming : A unified view of metaheuristics ». *European Journal of Operational Research*, 135 :1–16, 2001.

[66] Y. Toyoda. « A simplified algorithm for obtaining approximate solutions to zero-one programming problem ». *Management Science*, 21[12] :1417–1427, 1975.

[67] M. Vasquez et J. Hao. « An hybrid approach for the 0-1 multidimensional knapsack problem. ». *Proc. 17th International Joint Conference on Artificial Intelligence, IJCAI-01, Seattle, WA*, August 2001.

[68] M. Vasquez et Y. Vimont. « Improved results on the 0-1 multidimensional knapsack problem. ». *European Journal of Operational Research*, [165] :70–81, 2005.

[69] Y. Vimont, S. Boussier et M. Vasquez. « Reduced costs propagation in an efficient implicit enumeration for the 01 multidimensional knapsack problem ». *Journal Of Combinatorial Optimization*, 15[2] :165–178, 2008.

[70] H.M WEIGARTNER et D.N NESS. « Methods for the solution of the multidimensional 0/1 knapsack problem ». *Operations research*, 15 :83–103, 1967.

[71] H.M. WEINGARTNER. *Mathematical Programming and the analysis of Capital Budgeting Problems*. Prentice-Hall, 1963.

[72] C. WILBAUT et S. HANAFI. « New convergent heuristics for 0-1 mixed integer programming ». *European Journal of Operational Research*, doi :10.1016/j.ejor.2008.01.044., 2008.

[73] C. WILBAUT, S. HANAFI, A. FRÉVILLE et S. BALEV. « Tabu Search : Global Intensification using Dynamic Programming ». *Control and Cybernetics*, 35[3] :579–598, 2006.

[74] C. WILBAUT, S. HANAFI et S. SALHI. « A survey of effective heuristics and their application to a variety of knapsack problems ». *IMA Journal of Management Mathematics*, doi : 10.1093/imaman/dpn004, 2008.

[75] J. XU et S.Y. CHIU. « Effective heuristic procedures for a field technician scheduling problem ». *Journal of Heuristics*, 7 :495–509, 2001.

Titre : Étude de Resolution Search pour la Programmation Linéaire en Variables Binaires

Résumé : Dans cette thèse, nous nous intéressons à la résolution exacte de programmes linéaires en variables binaires. L'ensemble de nos travaux s'articule autour de l'étude de *Resolution search* (Chvátal (1997)) pour la résolution du problème du sac à dos multidimensionnel en 0–1. Dans un premier temps, nous proposons un algorithme d'énumération implicite centré sur une analyse des coûts réduits à l'optimum de la relaxation continue ainsi que sur une décomposition de l'espace de recherche en hyperplans. Nous proposons une stratégie de branchement originale visant à élaguer au plus tôt l'arbre de recherche. Cette stratégie est efficace pour résoudre des instances jugées difficiles mais rend l'algorithme dépendant de la connaissance d'une bonne solution de départ. Dans un deuxième temps, nous proposons une méthode de résolution plus autonome combinant Resolution search avec une énumération implicite inspirée du premier algorithme. Cette coopération permet d'obtenir rapidement de bonnes solutions et prouve les optimums d'instances de plus grande taille. Finalement, nous présentons une application de Resolution Search à la résolution d'un problème de planification dans le domaine des télécommunications.

Mots clés : Programmation linéaire en variables binaires, Resolution search, Backtracking intelligent, Enumération implicite, Sac à dos multidimensionnel en 0-1

Title: Study of Resolution Search for 0–1 Integer Linear Programming

Abstract: In this thesis, we are interested in the exact resolution of 0–1 integer linear programming problems. Our work revolves around the study of Resolution search (Chvátal (1997)) for solving the 0–1 multidimensional knapsack problem. As a first step, we propose an implicit enumeration algorithm based on an analysis of the reduced costs at the optimality of the problem's LP–relaxation and on the decomposition of the search space in hyperplanes. We propose an original branching strategy which focuses on pruning the search tree as soon as possible. This strategy is effective for solving difficult instances, however, it makes the algorithm depends on to the knowledge of a good starting solution. In a second step, we propose an exact method combining Resolution search with an implicit enumeration similar to the first algorithm. The resulting cooperation enables to obtain good solutions rapidly and to prove the optimality of several larger instances. Finally, we present an application of Resolution Search to a scheduling problem in the field of telecommunications.

Keywords: 0–1 Integer linear programming, Resolution search, Intelligent backtracking, Implicit enumeration, 0–1 Multidimensional knapsack problem

---

LGI2P, École des mines d'Alès
