“ Resolution Search ” et problèmes d’optimisation discrète

Marius Posta

▶ **To cite this version:**

Marius Posta. “ Resolution Search ” et problèmes d’optimisation discrète. Autre [cs.OH]. Université d’Avignon; Université de Montréal (1978-..), 2012. Français. NNT : 2012AVIG0189 . tel-00968201

HAL Id: tel-00968201

https://theses.hal.science/tel-00968201v1

Submitted on 31 Mar 2014

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

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

Université de Montréal
Université d'Avignon

« Resolution Search » et problèmes d'optimisation discrète

par
Marius Posta

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

et

Laboratoire d'Informatique d'Avignon
Université d'Avignon

Thèse présentée à la Faculté des arts et des sciences
en vue de l'obtention du grade de Philosophiæ Doctor (Ph.D.)
en informatique

Novembre, 2011

© Marius Posta, 2011.

Université de Montréal
Faculté des arts et des sciences

Cette thèse intitulée:

« Resolution Search » et problèmes d'optimisation discrète

présentée par:

Marius Posta

a été évaluée par un jury composé des personnes suivantes:

Jean-Yves Potvin,	président-rapporteur
Jacques A. Ferland,	directeur de recherche
Philippe Michelon,	codirecteur
Bernard Gendron,	membre du jury
Vašek Chvátal,	examinateur externe
Fred Glover,	examinateur externe
Jacques Bélair,	représentant du doyen de la FAS

Thèse acceptée le 3 Février 2012

RÉSUMÉ

Les problèmes d'optimisation discrète sont pour beaucoup difficiles à résoudre, de par leur nature combinatoire. Citons par exemple les problèmes de programmation linéaire en nombres entiers. Une approche couramment employée pour les résoudre exactement est l'approche de Séparation et Évaluation Progressive. Une approche différente appelée « Resolution Search » a été proposée par Chvátal en 1997 pour résoudre exactement des problèmes d'optimisation à variables 0-1, mais elle reste mal connue et n'a été que peu appliquée depuis.

Cette thèse tente de remédier à cela, avec un succès partiel. Une première contribution consiste en la généralisation de Resolution Search à tout problème d'optimisation discrète, tout en introduisant de nouveaux concepts et définitions. Ensuite, afin de confirmer l'intérêt de cette approche, nous avons essayé de l'appliquer en pratique pour résoudre efficacement des problèmes bien connus. Bien que notre recherche n'ait pas abouti sur ce point, elle nous a amené à de nouvelles méthodes pour résoudre exactement les problèmes d'affectation généralisée et de localisation simple. Après avoir présenté ces méthodes, la thèse conclut avec un bilan et des perspectives sur l'application pratique de Resolution Search.

Mots clés: Resolution Search, optimisation discrète, affectation généralisée, localisation simple.

ABSTRACT

The combinatorial nature of discrete optimization problems often makes them difficult to solve. Consider for instance integer linear programming problems, which are commonly solved using a Branch-and-Bound approach. An alternative approach, Resolution Search, was proposed by Chvátal in 1997 for solving 0-1 optimization problems, but remains little known to this day and as such has seen few practical applications.

This thesis attempts to remedy this state of affairs, with partial success. Its first contribution consists in the generalization of Resolution Search to any discrete optimization problem, while introducing new definitions and concepts. Next, we tried to validate this approach by attempting to solve well-known problems efficiently with it. Although our research did not succeed in this respect, it lead us to new methods for solving the generalized assignment and uncapacitated facility location problems. After presenting these methods, this thesis concludes with a summary of our attempts at practical application of Resolution Search, along with further perspectives on this matter.

Keywords: Resolution Search, discrete optimization, generalized assignment, uncapacitated facility location.

TABLE DES MATIÈRES

RÉSUMÉ . . . . . **iii**

ABSTRACT . . . . . **iv**

TABLE DES MATIÈRES . . . . . **v**

LISTE DES TABLEAUX . . . . . **viii**

LISTE DES FIGURES . . . . . **ix**

REMERCIEMENTS . . . . . **x**

INTRODUCTION **1**

CHAPITRE 1 : GENERALIZED RESOLUTION SEARCH **8**

1.1 Introduction and general concepts 9

1.1.1 Example 10

1.1.2 Definitions 12

1.2 Outline of the Resolution Search approach 13

1.3 Path-like structure 15

1.4 Updating the family 18

1.4.1 S contains at least one markable predicate for $U_{\mathcal{F}}$ 20

1.4.2 S contains no markable predicate for $U_{\mathcal{F}}$ 22

1.5 Complete algorithm and convergence 27

1.6 Concluding remarks 30

1.6.1 Recycling nogood clauses 30

1.6.2 Example of a possible application 30

1.6.3 Acknowledgements 32

1.7 Appendix 32

CHAPITRE 2 : AN EXACT METHOD WITH VARIABLE FIXING FOR SOLVING THE GENERALIZED ASSIGNMENT PROBLEM 37

2.1 Introduction 38
2.2 Solving a sequence of decision problems 40
2.3 Variable fixing 42
2.4 Lagrangian reduced costs 44
2.4.1 Decomposition of the lagrangian relaxation 44
2.4.2 Dynamic programming approach 45
2.5 Computational experience 48
2.5.1 Implementation details 49
2.5.2 Experimental protocol 50
2.5.3 Results 51
2.5.4 Concluding analysis 52

CHAPITRE 3 : AN EXACT COOPERATIVE METHOD FOR THE UNCAPACITATED FACILITY LOCATION PROBLEM 59

3.1 Introduction 60
3.2 Primal process 62
3.3 Dual process 66
3.3.1 Subproblems and lagrangian relaxation 66
3.3.2 Branch-and-bound procedure and dual process 70
3.3.3 Reoptimization 77
3.3.4 Bundle Method 83
3.4 Computational Results 88
3.4.1 Concurrency 89
3.4.2 Parameters 89
3.4.3 Tests 90
3.4.4 Solving UflLib 91

3.5 Concluding remarks 95
3.6 Appendices 97
3.6.1 Bundle method and cache implementation details 97
3.6.2 Computational results 100

CHAPITRE 4 : COMPTE-RENDU – RESOLUTION SEARCH EN TANT QU'ALTERNATIVE AU BRANCH-AND-BOUND 104

4.1 Aperçu général de `obstacle` avec recyclage de clauses 104
4.2 Cas du problème d'affectation généralisé 109
4.3 Cas du problème de localisation simple 112
4.4 Bilan 117

CONCLUSION 120

BIBLIOGRAPHIE 123

LISTE DES TABLEAUX

2.I Solution values found by our method. 57

2.II Performance. 58

3.I Solving with `DFS` and $|\mathcal{C}| < 128$. 94

3.II Solving with `BFS` and no cache. 95

3.III Optimal values for the Körkel-Ghosh instance set 102

3.IV Optimal values for the M* instance set. 103

LISTE DES FIGURES

1.1 A problem instance, and a solution $(\tilde{l}, \tilde{h}) = (3, 2)$. 11
1.2 $S = \texttt{obstacle}(U_{\mathcal{F}})$ maintains the path-like structure for $U_{\mathcal{F}}$. . 20
1.3 Search state at this iteration. 21
1.4 Search state after this iteration. 22
1.5 Search state at this iteration. 25
1.6 Search state after this iteration. 26

2.1 The graph $G^i(\boldsymbol{\lambda})$. 46
2.2 The new shortest path after fixing $\mathbf{x}_{ij'}$ to 1. 48
2.3 Impact of variable fixing on the time required for instance e10100. 53
2.4 Evolution of bound values for instance d05100. 55

3.1 Preprocessing and branching settings. 92
3.2 Cache settings. 93
3.3 Profile curves for `UflLib`. 96

REMERCIEMENTS

Je voudrais avant tout remercier mes directeurs de recherche, les Professeurs Ferland et Michelon, pour leur soutien sans lequel cette thèse n'aurait pas été. Je les remercie également pour les encouragements et la confiance qu'ils m'ont témoigné durant toutes ces années de travail.

Je remercie la communauté académique avec laquelle j'ai été en contact pour avoir contribué à m'enrichir intellectuellement, par le biais de cours, de conférences, de séminaires, et surtout des conversations stimulantes. Les intéressés se reconnaîtront sans doute et j'espère les retrouver au McCarold pour une bière ou deux.

Finalement, je remercie ma famille pour son soutien.

INTRODUCTION

Les problèmes d'optimisation discrète sont pour beaucoup difficiles à résoudre exactement, de par leur nature combinatoire. Citons par exemple les problèmes de programmation linéaire en nombres entiers. Une approche couramment employée pour les résoudre exactement est l'approche de Séparation et Évaluation Progressive (*Branch-and-bound*). Une approche différente, appelée Resolution Search, a été proposée par Chvátal [11] en 1997 pour résoudre exactement des problèmes d'optimisation à variables 0-1, mais elle reste mal connue et n'a été que peu appliquée depuis.

Tout comme le Branch-and-bound, Resolution Search applique le principe de « diviser-pour-régner » pour résoudre un problème d'optimisation discrète, que l'on peut représenter de manière générale par

$$\min \{ f(\mathbf{x}) : \mathbf{x} \in \mathcal{X} \},$$

où $\mathcal{X}$ est le domaine des variables de décisions, et où f est une fonction objectif qui renvoie le coût d'une solution entière $\mathbf{x} \in \mathcal{X}$ si celle-ci est réalisable, et $+\infty$ sinon. Dénotons également la meilleure solution connue par $\mathbf{x}^*$ et sa valeur par $\bar{z}$. Cette valeur constitue une borne supérieure à la valeur optimale du problème. L'approche Branch-and-bound consiste à partitionner $\mathcal{X}$ récursivement et à éliminer les parties pour lesquelles nous savons qu'elles ne contiennent pas de solution meilleure que $\mathbf{x}^*$. Ce constat peut être fait pour une partie $X \subseteq \mathcal{X}$ en calculant une borne inférieure sur le coût d'une solution dans X, tout en maintenant $\mathbf{x}^*$ et $\bar{z}$ à jour. Si cette borne inférieure égale ou excède la borne supérieure $\bar{z}$, alors nous n'avons pas besoin de partitionner X davantage, sinon nous partitionnons X en deux parties non vides et répétons la procédure.

Prenons par exemple un problème de programmation linéaire mixte avec va-

riables binaires quelconque. Celui-ci peut être modélisé sous la forme suivante :

$$\begin{aligned} \min \quad & \mathbf{c}^T\mathbf{x} + \mathbf{d}^T\mathbf{y} & \\ \text{s.à} \quad & \mathbf{A}\mathbf{x} + \mathbf{E}\mathbf{y} = \mathbf{b}, & (1)\\ & \mathbf{0} \leq \mathbf{x} \leq \mathbf{1}, & (2)\\ & \mathbf{x} \text{ entier}, & (3)\\ & \mathbf{y} \geq \mathbf{0}. & (4) \end{aligned}$$

Nous avons ici $\mathcal{X} = \{0,1\}^n$ et

$$\begin{aligned} f : \mathcal{X} &\longrightarrow \mathbb{R} \cup \{+\infty\} \\ \mathbf{x} &\longmapsto \mathbf{c}^T\mathbf{x} + \inf_{\mathbf{y}\geq\mathbf{0}} \left\{\mathbf{d}^T\mathbf{y} : \mathbf{E}\mathbf{y} = \mathbf{b} - \mathbf{A}\mathbf{x}\right\}. \end{aligned}$$

Une borne inférieure pour tout sous-ensemble $E \subseteq \mathcal{X}$ peut alors être obtenue en calculant $\min_{\mathbf{x}\in conv(E)} f(\mathbf{x})$, où $conv(E)$ désigne l'enveloppe convexe de E. Typiquement, une résolution par Branch-and-bound implique que nous partitionnons $\mathcal{X}$ en fixant certaines composantes des solutions $\mathbf{x}$ à 0 ou à 1. Dans ce cas, une solution partielle $\mathbf{p} \in \{0,1,*\}^n$ caractérise n'importe quelle partie de $\mathcal{X}$ ainsi obtenue, et nous avons

$$X(\mathbf{p}) = \{\mathbf{x} \in \mathcal{X} : \forall i \in \{1,\ldots,n\}, \mathbf{p}_i \in \{0,1\} \implies \mathbf{x}_i = \mathbf{p}_i\}.$$

Nous obtenons alors une borne inférieure en résolvant le programme linéaire $LP(\mathbf{p})$:

$$\begin{aligned} \min \quad & \mathbf{c}^T\mathbf{x} + \mathbf{d}^T\mathbf{y} & \\ \text{s.à} \quad & \mathbf{A}\mathbf{x} + \mathbf{E}\mathbf{y} = \mathbf{b}, & (1)\\ & \mathbf{0} \leq \mathbf{x} \leq \mathbf{1}, & (2)\\ & \mathbf{x}_i = \mathbf{p}_i, \qquad \text{si } \mathbf{p}_i \in \{0,1\}, \forall i \in \{1,\ldots,n\}, & (3')\\ & \mathbf{y} \geq \mathbf{0}. & (4) \end{aligned}$$

Si nous voyons le Branch-and-bound comme une approche de haut en bas (*top-down*), alors Resolution Search est au contraire une approche de bas en haut (*bottom-up*), qui consiste à identifier des sous-ensembles de $\mathcal{X}$ ne contenant que des solutions coûtant au moins $\bar{z}$ et à les agréger ensemble. Resolution Search, telle que proposée par Chvátal [11] pour le cas où $\mathcal{X} = \{0,1\}^n$, maintient un état de la recherche qui recense des sous-ensembles de $\mathcal{X}$ déjà explorés, dans le sens où nous connaissons le coût minimum d'une solution dans l'union de ceux-ci. À partir de celui-ci, Resolution Search définit une solution partielle $\mathbf{u} \in \{0,1,*\}^n$ telle que $X(\mathbf{u})$ soit disjoint des sous-ensembles de $\mathcal{X}$ déjà explorés. Pour poursuivre la résolution du problème, nous devons appliquer une procédure `obstacle` définie spécifiquement pour le problème en question et qui génère une solution partielle $\mathbf{s} \in \{0,1,*\}^n$ telle que $X(\mathbf{s})$ ait été exploré (i.e. telle que $\min f(X(\mathbf{s}))$ soit connu et $\bar{z}$ et $\mathbf{x}^*$ mis à jour) et telle que $X(\mathbf{s}) \cap X(\mathbf{u})$ soit non vide. Resolution Search met ensuite à jour son état de la recherche avec $\mathbf{s}$ de telle sorte que d'itération en itération, tout le domaine $\mathcal{X}$ ait été exploré.

Un exemple de procédure `obstacle` pour la programmation linéaire mixte avec variables 0-1 a été abordé dans [11], et consiste à faire les choses suivantes, étant donné $\mathbf{u} \in \{0,1,*\}^n$.

0. Initialiser $\mathbf{p} \leftarrow \mathbf{u}$.

1. Si $LP(\mathbf{p})$ est irréalisable, ou si la valeur optimale de $LP(\mathbf{p})$ égale ou dépasse $\bar{z}$, aller à l'étape 3.

2. Choisir un indice $i \in \{1,\ldots,n\}$ pour laquelle la composante $\mathbf{x}_i$ est fractionnaire à l'optimum de $LP(\mathbf{p})$. S'il n'y en a pas, mettre à jour $\mathbf{x}^*$ et $\bar{z}$ avec la solution et la valeur optimales de $LP(\mathbf{p})$, respectivement. Sinon, fixer $\mathbf{p}_i$ à 0 ou à 1 (au choix) et retourner à l'étape 1.

3. Initialiser $\mathbf{s} \leftarrow \mathbf{p}$ et relâcher des composantes de $\mathbf{s}$ de telle sorte que $LP(\mathbf{s})$ demeure soit irréalisable, soit avec une valeur optimale non inférieure à $\bar{z}$. Renvoyer $\mathbf{s}$.

Chvátal a remarqué que l'étape 3 pouvait se faire relativement efficacement en se servant de l'information duale obtenue lors de la résolution de la relaxation linéaire $LP(\mathbf{s})$. En effet, dans le cas où $LP(\mathbf{s})$ est réalisable, il est possible de relacher toute composante $\mathbf{s}_i$ fixée à 0 si la variable $\mathbf{x}_i$ correspondante a un coût réduit non négatif, et vice-versa pour $\mathbf{s}_i = 1$.

C'est à partir de ce genre d'observations que nous avons émis l'hypothèse que Resolution Search pouvait peut-être devenir une approche plus efficace pour résoudre des problèmes d'optimisation combinatoire : plus efficace dans le sens où elle permet d'exploiter plus amplement la structure du problème à résoudre que ne le permet le Branch-and-bound. L'objectif principal de cette thèse a été de confirmer cette hypothèse en identifiant un problème pour lequel une méthode du type Resolution Search nous donnerait de bons résultats. Un autre objectif était de mieux comprendre Resolution Search, qui était (et est encore) une approche méconnue et mal comprise. Entre autres choses, y-a-t'il un lien entre Resolution Search et d'autres approches comme les métaheuristiques ou la programmation par contraintes ?

Une première et importante étape de notre travail a été la généralisation de Resolution Search, présentée dans le chapitre 1. Ce chapitre consiste en un article depuis publié dans *Discrete Optimization* [44]. Ensuite, nous avons cherché à valider l'utilité de Resolution Search d'un point de vue pratique sur des problèmes académiques classiques : la programmation en nombres entiers notamment, mais aussi le problème d'ordonnancement d'ateliers, l'affectation généralisée, etc. Bien que les résultats de notre recherche subséquente ne nous aient pas permis cela, l'étude de ces problèmes a abouti à d'autres résultats intéressants par eux-mêmes, notamment pour le problème d'affectation généralisée et pour le problème de localisation simple.

Le problème d'affectation généralisé consiste à affecter chaque tâche $j \in \{1, \ldots, n\}$ à un et un seul des m agents dans l'objectif de minimiser les coûts des affectations. Par contre, plusieurs tâches peuvent être affectées à un seul agent $i \in \{1, \ldots, m\}$, mais sa disponibilité globale est limitée par une contrainte qui prend la forme d'une

inégalité knapsack : b_i unités de ressource sont disponibles, et l'affectation d'une tâche j en consomme a_{ij}. Le problème peut être modélisé de la manière suivante :

$$\begin{aligned} \min \quad & \sum_{i=1}^{m}\sum_{j=1}^{n} \mathbf{c}_{ij}\mathbf{x}_{ij} && (GAP) \\ \text{s.à} \quad & \sum_{j=1}^{n} \mathbf{a}_{ij}\mathbf{x}_{ij} \leq \mathbf{b}_i, && 1 \leq i \leq m, \ (1) \\ & \sum_{i=1}^{m} \mathbf{x}_{ij} = 1, && 1 \leq j \leq n, \ (2) \\ & \mathbf{x} \in \{0,1\}^{mn}. && (3) \end{aligned}$$

Il est bien entendu possible de résoudre (GAP) par Branch-and-bound, mais la bonne performance d'une telle méthode dépend étroitement de la découverte rapide d'une bonne solution réalisable, c'est-à-dire un vecteur $\mathbf{x}$ satisfaisant toutes les contraintes (1), (2) et (3). Ceci peut néanmoins s'avérer difficile selon les valeurs que prennent les coefficients des contraintes de ressource (1).

Dans le chapitre 2 contenant un article publié dans *Computational Optimization and Applications* [43], nous proposons une méthode qui contourne complètement cette difficulté en transformant ce problème d'optimisation en une suite de problèmes de décision. Un tel problème de décision est posé pour une borne supérieure artificielle $\bar{z}$ et consiste à déterminer l'existence ou non d'une solution réalisable de coût inférieur ou égal à $\bar{z}$. Comme les coefficients $\mathbf{a}$, $\mathbf{b}$ et $\mathbf{c}$ définissant une instance du problème sont entiers, nous prenons comme borne supérieure artificielle $\bar{z}$ initiale le plafond d'une borne inférieure globale et cherchons une solution réalisable. Si une telle solution existe, elle est nécessairement optimale. Sinon, nous incrémentons $\bar{z}$ de 1 et recommençons une nouvelle recherche, et ainsi de suite. La recherche est effectuée au moyen d'un Branch-and-bound utilisant la relaxation lagrangienne obtenue en dualisant les contraintes d'affectation (2), qui peut produire des bornes inférieures plus élevées que la relaxation linéaire. Afin de limiter le plus possible la duplication d'effort inhérente à ce genre d'approche, nous introduisons égale-

ment un mécanisme de fixation de variables qui contribue à rendre notre methode efficace. Nous avons ainsi obtenu une méthode exacte assez simple qui améliore sensiblement l'état de l'art. Celui-ci [2] consistait auparavant en une méthode de génération de coupes de couverture du polytope knapsack, c'est-à-dire l'enveloppe convexe de

$$\left\{ \mathbf{x} \in \{0,1\}^{mn} \ : \ \sum_{j=1}^{n} \mathbf{a}_{ij}\mathbf{x}_{ij} \leq \mathbf{b}_i, \ 1 \leq i \leq m \right\}.$$

La génération de coupes était appliquée avant de résoudre le problème par un Branch-and-bound tel CPLEX, dans le but de renforcer la relaxation linéaire.

Le problème de localisation simple consiste à décider pour n différents endroits de l'ouverture d'un dépot ou non, afin d'approvisionner m clients à moindre coût. L'approvisionnement d'un client j à partir d'un dépôt situé à l'endroit i entraîne un coût $\mathbf{s}_{ij}$, et l'ouverture d'un dépôt en i entraine un coût $\mathbf{c}_i$; si $\mathbf{x} \in \{0,1\}^n$ désigne une solution telle que les composantes à 1 correspondent aux endroits où l'on décide d'ouvrir un dépôt, alors $\mathbf{x}$ coûte

$$\sum_{i=1}^{n} \mathbf{c}_i\mathbf{x}_i + \sum_{j=1}^{m} \min\left\{ \mathbf{s}_{ij} : \mathbf{x}_i = 1, \ 1 \leq i \leq n \right\}.$$

Dans ce cas également, le problème de trouver une solution de coût minimum peut être modélisé par un programme linéaire avec variables binaires :

$$\begin{aligned}
\min \quad & \sum_{i=1}^{n} \mathbf{c}_i\mathbf{x}_i + \sum_{j=1}^{m}\sum_{i=1}^{n} \mathbf{s}_{ij}\mathbf{y}_{ij} & & \\
\text{s.à} \quad & \sum_{i=1}^{n} \mathbf{y}_{ij} = 1, & 1 \leq j \leq m, & \quad (1) \\
& 0 \leq \mathbf{y}_{ij} \leq \mathbf{x}_i, & 1 \leq i \leq n, 1 \leq j \leq m, & \quad (2) \\
& \mathbf{x} \in \{0,1\}^n. & & \quad (3)
\end{aligned}$$

Ce problème peut donc être résolu par Branch-and-bound de manière standard, en appliquant CPLEX par exemple. En pratique cependant, ceci peut s'avérer

difficile du fait que la relaxation linéaire utilisée est fortement dégénérée et requiert beaucoup d'itérations du simplexe pour être résolue exactement. Une méthode exacte assez ancienne (`DualLoc`) pallie à cette difficulté en résolvant la relaxation linéaire de manière approchée en appliquant une heuristique *dual ascent*.

Le chapitre 3, qui correspond au troisième article de cette thèse, propose une nouvelle méthode exacte qui prend une voie intermédiaire pour le Branch-and-bound en résolvant une relaxation lagrangienne de manière approchée par une méthode de faisceaux (*bundle search method*). De plus, l'espace de solutions est séparé non seulement en branchant sur la construction ou non d'un dépôt à un endroit donné (c'est-à-dire en fixant une composante de $\mathbf{x}$), mais aussi en branchant sur le nombre de dépôts à ouvrir (c'est à dire en restreignant le résultat de la somme $\sum_{i=1}^{n} \mathbf{x}_i$). D'une manière similaire à notre travail sur le problème d'affectation généralisé, nous introduisons des règles de fixation de variable qui améliorent l'efficacité du Branch-and-bound. Enfin, notre méthode est une méthode coopérative alliant une recherche métaheuristique du type tabou à une recherche exacte par Branch-and-bound. Ces deux processus de recherche peuvent être exécutés parallèlement et coopèrent en s'échangeant de l'information. Notre méthode donne de bonnes performances sur des instances de nature très différente, et résout à l'optimalité plusieurs instances qui étaient jusqu'ici hors de portée tant de `DualLoc` que de solveurs comme CPLEX.

Enfin, le chapitre 4 constitue une annexe aux trois articles de cette thèse, et résume en quelque sorte nos tentatives d'application pratique de Resolution Search, notamment aux problèmes traités dans les deux derniers articles. Bien que ces pistes n'aient pas abouti, elles peuvent éventuellement ouvrir des avenues de recherche futures. Nous y présentons également une généralisation d'une technique qui permet potentiellement d'améliorer l'application de Resolution Search en stockant et réutilisant toute les informations recueillies sur l'espace de solutions $\mathcal{X}$.

CHAPITRE 1

GENERALIZED RESOLUTION SEARCH

un article écrit par

Marius Posta[1,2], Jacques A. Ferland[1], Philippe Michelon[2]

et publié dans

Discrete Optimization, pages 215 à 228, volume 8, numéro 2, année 2011.

Abstract

Difficult discrete optimization problems are often solved using a Branch-and-Bound approach. Resolution Search is an alternate approach proposed by Chvátal for 0-1 problems, allowing more flexibility in the search process. In this paper, we generalize the Resolution Search approach to any discrete problem.

1. Département d'Informatique et de Recherche Opérationnelle,
   Université de Montréal,
   C.P. 6128, succursale Centre-ville,
   Montréal, Québec, Canada, H3C 3J7.

2. Laboratoire d'Informatique d'Avignon,
   Université d'Avignon et des Pays du Vaucluse,
   84911 Avignon, Cedex 9, France.

1.1 Introduction and general concepts

Consider an optimization problem where the set of feasible solutions $\mathcal{X}_{feas}$ is discrete, and where the goal is to find the element of $\mathcal{X}_{feas}$ which minimizes a real function f'. Very often the contents of $\mathcal{X}_{feas}$ are not known explicitly, because this set is specified by constraints on a discrete superset $\mathcal{X}$. This is the case in integer programming for example, where $\mathcal{X}$ is a hypercube of integer vectors and $\mathcal{X}_{feas}$ is specified as the subset of solutions in $\mathcal{X}$ which satisfy a number of problem-specific linear inequalities. In this paper, we formulate the problem as follows:

$$\min\{f(x) : x \in \mathcal{X}\} \tag{P}$$

where $\mathcal{X}$ is discrete and satisfies $\mathcal{X}_{feas} \subseteq \mathcal{X}$, and where the objective function f is defined as follows:

$$\begin{array}{rccl} f: & \mathcal{X} & \longrightarrow & \mathbb{R} \cup \{+\infty\} \\ & x & \longmapsto & \begin{cases} f'(x) \text{ if } x \in \mathcal{X}_{feas}, \\ +\infty \text{ otherwise.} \end{cases} \end{array}$$

Solving (P) consists in identifying a solution $x^* \in \mathcal{X}$ which minimizes the objective function f: if $f(x^*) = +\infty$, then $\mathcal{X}_{feas}$ is empty, otherwise x^* minimizes f' in $\mathcal{X}_{feas}$. This paper shows how this may be done using a Resolution Search approach, which is an alternative to the popular Branch-and-Bound approach.

Chvátal was the first author to introduce a Resolution Search approach to solve binary variable problems (i.e. those where $\mathcal{X} = \{0,1\}^n$) in [11]. The approach relies on a specific procedure `obstacle` which, in essence, identifies subsets of $\mathcal{X}$ in which no solution is better than the current best known. A few straightforward implementation attempts were then published, where a Resolution Search approach was used to deal with problems with varying degrees of success. Demassey et al. [13] were the first to implement a Resolution Search approach, which they applied to

solve a scheduling problem. Then Palpant et al. [40], and more recently Boussier et al. [10], applied the approach to deal with the n^2-queens and 0-1 multidimensional knapsack problems, respectively. These authors focused on finding an application for which a Resolution Search approach would be competitive compared to the more traditional Branch-and-Bound approach, and they did not develop the theoretical aspects of Resolution Search much further. Demassey et al. [13], and Palpant et al. [40] report encouraging results despite not being competitive with the state of the art for their respective problems. Furthermore, Boussier et al. [10] were able to exactly solve previously unsolved instances of the 0-1 multidimensional knapsack problem.

Hanafi and Glover [25] generalized the Resolution Search approach by extending Chvátal's specific procedure `obstacle` to mixed integer programs and adapting the search procedure accordingly. Furthermore, they provided interesting parallels with earlier approaches like Dynamic Branch-and-Bound. The contributions of this paper are to further generalize the Resolution Search approach to any discrete optimization problem (P), and to formally prove convergence of the search procedure with a minimum of hypotheses on an otherwise unspecified `obstacle`. For this purpose, we lean heavily on the concepts introduced in [11]. Since they were presented in the less general context of 0-1 problems, we have to redefine some of them in addition to defining new ones.

1.1.1 Example

We now introduce a toy problem and an instance which will be used as an example throughout this paper. Consider a decreasing curve Γ in the plane, and suppose that its mathematical formulation is not known explicitely. However, we know where Γ intersects the coordinate axes, and for any point (l, h) an oracle is available to indicate if it lies below or above Γ. The objective is to determine the rectangle with the largest area while satisfying the following constraints:

- its extreme points must be $(0, 0)$, $(0, h)$, $(l, 0)$ and (l, h),

- l and h must be non-negative integers,
- and (l, h) must lie below Γ.

Assume that Γ intersects the coordinate axes at $(l_\cap, 0)$ and $(0, h_\cap)$. If we denote $l_{\max} = \lfloor l_\cap \rfloor$ and $h_{\max} = \lfloor h_\cap \rfloor$, then the search space for this problem is:

$$\mathcal{X} = \left\{(l, h) \in \mathbb{Z}^2 : 0 \leq l \leq l_{\max}, 0 \leq h \leq h_{max}\right\}.$$

Furthermore, denote the objective function f to be minimized over $\mathcal{X}$ as follows:

$$f(l, h) = \begin{cases} +\infty \text{ if } (l, h) \text{ lies above } \Gamma \\ -lh \text{ otherwise.} \end{cases}$$

An initial upper bound on the optimal value is $\bar{z} = 0$. Denote by $r^* = (l^*, h^*)$ the best known solution so far.

In Figure 1.1, the curve Γ intersects the l axis between 5 and 6 and the h axis between 4 and 5. Accordingly, given that Γ is decreasing it follows that the optimal upper right corner of the rectangle must belong to the following set:

$$\mathcal{X} = \{(l, h) \in \mathbb{Z}^2 : 0 \leq l \leq 5, 0 \leq h \leq 4\}.$$

In the figure, the point $(\tilde{l}, \tilde{h}) = (3, 2)$ lies below the curve so that $f(3, 2) = -6$.

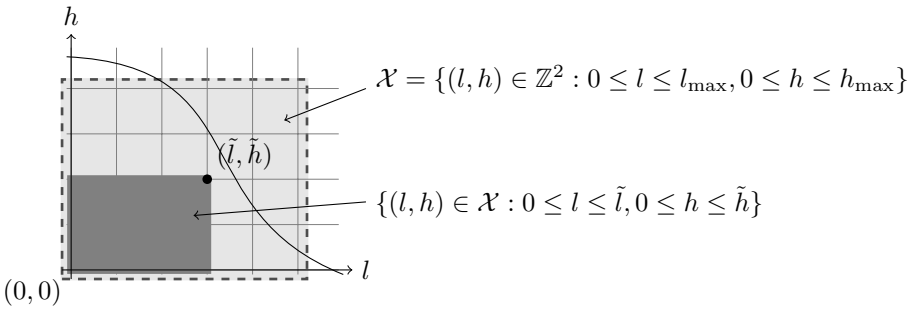


Figure 1.1: A problem instance, and a solution $(\tilde{l}, \tilde{h}) = (3, 2)$.

1.1.2 Definitions

In general, a problem of type (P) is too difficult to be solved directly. For this reason we often use the popular divide-and-conquer strategy: the solution set $\mathcal{X}$ is partitioned into subsets $\mathcal{X}_1 \cup \mathcal{X}_2 \cup ... \cup \mathcal{X}_m = \mathcal{X}$ such that the objective function f is easily minimized on each subset. Then a minimum value z^* of f can be inferred on $\mathcal{X}$. Note that if we consider the parts of $\mathcal{X}$ successively one after the other, as we usually do, it is not always necessary to find a minimum on each of them. Indeed, assume that $\bar{z}$ is the value of the best known solution when the subset $\mathcal{X}_i$ is to be examined. If we can verify that the mimimum value of f on $\mathcal{X}_i$ is greater than or equal to $\bar{z}$, then we can infer that no better solution of the problem (P) can be found in $\mathcal{X}_i$.

First we introduce some preliminary definitions.

Definition 1. We declare that a subset $\mathcal{X}_i$ of $\mathcal{X}$ has been **explored** when we know that this subset does not contain any solution with a smaller objective value than the best known in $\mathcal{X}$.

Hence the problem (P) is solved once $\mathcal{X}$ has been entirely explored.

Definition 2. A set C of predicates on $\mathcal{X}$ is called a **clause**. It defines a subset $X(C)$ of $\mathcal{X}$. $X(C)$ is called its **clause cover**.

$$X(C) = \{x \in \mathcal{X} : x \text{ satisfies all predicates } \gamma \in C\}$$

Any solution or any set of solutions is said to be **covered** by a clause C if it is included in the clause cover $X(C)$. For the sake of consistency, $X(\emptyset) = \mathcal{X}$. Furthermore, we assume that any predicate γ included in any clause has a **complement predicate** $\bar{\gamma}$ in the sense that any $x \in \mathcal{X}$ satisfies either γ or $\bar{\gamma}$.

Here we merely extend the concept of clauses, as presented by Chvátal [11] for 0-1 problems, and subsequently generalized to integer programming by Hanafi and Glover [25]. Chvátal [11] defines his clauses as sets of literals, and literals can be seen as a particular kind of predicate.

Example 1. As an example, consider the set $\mathcal{X} = \{(l,h) \in \mathbb{Z}^2 : 0 \leq l \leq 5, 0 \leq h \leq 4\}$ of our toy problem. Let us define a predicate γ, satisfied for any $(l,h) \in \mathcal{X}$ if and only if $0 \leq l \leq 3$. The predicate $\bar{\gamma}$ satisfied for any $(l,h) \in \mathcal{X}$ such that $4 \leq l \leq 5$ is complementary to γ. □

The notation $X(C)$ to designate the set of solutions satisfying a clause C is borrowed from Hanafi and Glover [25].

Remark 1. Let A and B be two clauses. Then $X(A \cup B) = X(A) \cap X(B)$, and $A \subseteq B$ implies $X(B) \subseteq X(A)$.

Definition 3. A clause C is declared **nogood** if the corresponding cover $X(C)$ has been explored. Note that the word 'nogood' is borrowed from the constraint programming terminology. Chvátal [11] would say that a specific clause is '$\bar{z}$-forcing' instead.

1.2 Outline of the Resolution Search approach

In the Branch-and-Bound approach, the search space $\mathcal{X}$ is partitioned recursively. Let $\mathcal{L}$ be the list containing the subsets of $\mathcal{X}$ left to be explored (initially, $\mathcal{L} = [\mathcal{X}]$). An upper bound $\bar{z}$ on the optimal value of (P) corresponding to the best known solution is available, and it is updated during the procedure. At a specific iteration i, a subset $\mathcal{X}_i \subseteq \mathcal{X}$ is removed from the list $\mathcal{L}$. A bounding procedure is applied on $\mathcal{X}_i$ to generate a lower bound $\underline{z}_i$ on the values in $f(\mathcal{X}_i)$. In doing so the bounding procedure may also update an upper bound $\bar{z}$ on the optimal value of (P). If $\underline{z}_i < \bar{z}$, then the subset $\mathcal{X}_i$ is split into at least two subsets to be added to $\mathcal{L}$. The procedure then completes the next iteration $i+1$, unless $\mathcal{L}$ is empty, in which case $\bar{z}$ is the optimal value of (P).

The Resolution Search approach does not rely on such a bounding procedure. Instead, at each iteration of the Resolution Search, a clause U specifies a nonempty subset $X(U) \subseteq \mathcal{X}$ which is to be at least partly explored using a procedure `obstacle`. This procedure generates a nogood clause S which covers some ele-

ments of $X(U)$ (i.e. $X(S) \cap X(U) \neq \emptyset$), and also updates the upper bound $\bar{z}$ on the optimal value of (P).

Note that in order to implement efficiently the Resolution Search approach for solving an optimization problem, the definition of the `obstacle` procedure is a key point, in the same way as computing a lower bound is a matter of prime importance in a Branch-and-Bound scheme.

Example 2. For our toy problem, a procedure `obstacle` can be specified as follows. Consider the clause U, and select $(\tilde{l}, \tilde{h})$ arbitrarily among the elements in $X(U)$:

- If $(\tilde{l}, \tilde{h})$ lies above Γ, then any solution (l, h) satisfying $\tilde{l} \leq l \leq l_{\max}$ and $\tilde{h} \leq h \leq h_{\max}$ has a value $f(l, h) = +\infty$.
- If $(\tilde{l}, \tilde{h})$ lies below Γ, then any solution (l, h) satisfying $0 \leq l \leq \tilde{l}$ and $0 \leq h \leq \tilde{h}$ is feasible and the area of the corresponding rectangle is smaller than or equal to the one corresponding to $(\tilde{l}, \tilde{h})$.

Accordingly, the nogood clause is specified as $S = \{\tilde{l} \leq l \leq l_{\max}, \tilde{h} \leq h \leq h_{\max}\}$ or $S = \{0 \leq l \leq \tilde{l}, 0 \leq h \leq \tilde{h}\}$. Since $(\tilde{l}, \tilde{h}) \in X(U)$, it follows that $X(S) \cap X(U) \neq \emptyset$. Finally, this procedure updates $\bar{z}$ and $r^* = (l^*, h^*)$ when required. □

The nogood clauses generated by `obstacle` during the search are kept in a stack of clauses $\mathcal{F}$ called a **family**. Before outlining how Resolution Search explores $\mathcal{X}$ using a procedure `obstacle` in algorithm 1, we extend the notion of cover to families of clauses.

Definition 4. The **reach** $\mathcal{R}(\mathcal{F})$ of a family $\mathcal{F} = [C_1, C_2, ..., C_m]$ is the set of all solutions in $\mathcal{X}$ covered by at least one clause in $\mathcal{F}$:

$$\mathcal{R}(\mathcal{F}) = \bigcup_{i=1}^{m} X(C_i).$$

Note that the notion of reach is equivalent to the τ function of Chvátal [11], since $|\mathcal{R}(\mathcal{F})| = \tau(\mathcal{F})$ for 0-1 problems.

Algorithm 1 - Procedure `ResolutionSearch`:

Step 0. Let $\mathcal{F} = \emptyset$, the clause $U_{\mathcal{F}} = \emptyset$ and the integer $m = 0$.

Step 1. Let S be the nogood clause generated by `obstacle`$(U_{\mathcal{F}})$.

Step 2. Generate $\mathcal{F}'$ using $\mathcal{F}$ and the clause S. If $\mathcal{R}(\mathcal{F}') = \mathcal{X}$, then $\mathcal{X}$ has been completely explored. The search is then completed, and the optimal value of f on $\mathcal{X}$ is $\bar{z}$. Otherwise, generate a clause $U_{\mathcal{F}'}$, having a nonempty cover $X(U_{\mathcal{F}'})$ but sharing no elements with $\mathcal{R}(\mathcal{F}')$, i.e. $X(U_{\mathcal{F}'}) \cap \mathcal{R}(\mathcal{F}') = \emptyset$.

Step 3. Increment m, replace $\mathcal{F}$ by $\mathcal{F}'$ and $U_{\mathcal{F}}$ by $U_{\mathcal{F}'}$. Return to Step 1.

The main feature of the Resolution Search is included in Step 2. A naive way of implementing Step 2 would be as follows: generate $\mathcal{F}'$ by adding the clause S to the existing family $\mathcal{F}$. Assuming that it would be easy to generate a suitable clause $U_{\mathcal{F}'}$, then the reach of the family would grow strictly at each iteration (i.e. $\mathcal{R}(\mathcal{F}) \subsetneq \mathcal{R}(\mathcal{F}')$). Indeed, the nogood clause S has a cover $X(S)$ sharing some elements with $X(U_{\mathcal{F}})$, but $X(U_{\mathcal{F}})$ does not share any element with $\mathcal{R}(\mathcal{F})$. It follows that $\mathcal{X}$ would be explored in at most $|\mathcal{X}|$ iterations. However, since the family would grow as the search progresses, the generation of a suitable $U_{\mathcal{F}'}$ would become a problem in itself.

For this reason we impose an additional property on `obstacle`. It will allow the family $\mathcal{F}$ to maintain a special recursive structure, to generate more easily $U_{\mathcal{F}'}$ at each iteration. This structure on the family is defined in the following section.

1.3 Path-like structure

We first introduce the following definitions.

Definition 5. A predicate γ is said to be **markable** for a clause $U_{\mathcal{F}}$ if:

- γ partitions the search space (i.e. $X(\{\gamma\}) \neq \emptyset$ and $X(\{\bar{\gamma}\}) \neq \emptyset$), and
- γ is not in the clause $U_{\mathcal{F}}$ (i.e. $\gamma \notin U_{\mathcal{F}}$).

Example 3. Consider the toy problem presented earlier. In this context, the clauses are defined with predicates specified in terms of lower or upper bounds on the integer decision variables l and h. For the sake of clarity, the predicates (which are functions) are specified by the corresponding bounds on the variables.

Suppose that $U_{\mathcal{F}} = \{3 \leq l \leq 5, 0 \leq h \leq 3, 0 \leq l \leq 4\}$. The predicates $(1 \leq h \leq 4)$ and $(2 \leq l \leq 5)$ are markable for this $U_{\mathcal{F}}$, whereas $(0 \leq h \leq 3)$ is not because it is in $U_{\mathcal{F}}$, nor is $(0 \leq l)$ because it does not partition the search space (the predicate is satisfied by all $(l, h) \in \mathcal{X}$). □

Definition 6. A clause C is said to **maintain the path-like structure** for a clause $U_{\mathcal{F}}$ if for all markable predicates $\gamma \in C$, the intersection of $X(U_{\mathcal{F}})$ and $X(\bar{C}_\gamma)$ is nonempty (i.e. $X(U_{\mathcal{F}}) \cap X(\bar{C}_\gamma) \neq \emptyset$), where $\bar{C}_\gamma = (C \setminus \{\gamma\}) \cup \{\bar{\gamma}\}$.

We now define a specific recursive structure for the clause families.

Definition 7. The family $\mathcal{F}' = [C_1, C_2, ..., C_m, C_{m+1}]$ is **path-like** if the family $\mathcal{F} = [C_1, C_2, ..., C_m]$ is path-like, and if there is a clause $U_{\mathcal{F}}$ such that:

- Its cover is nonempty, i.e. $X(U_{\mathcal{F}}) \neq \emptyset$.
- Its cover has empty intersection with the reach of $\mathcal{F}$; i.e. $X(U_{\mathcal{F}}) \cap \mathcal{R}(\mathcal{F}) = \emptyset$.
- The clause C_{m+1} contains at least one markable predicate for $U_{\mathcal{F}}$.
- The clause C_{m+1} maintains the path-like structure for $U_{\mathcal{F}}$.

In order to use the recursivity of this definition, we consider an empty family $\mathcal{F}_0$ to be path-like. We assume that $\mathcal{R}(\mathcal{F}_0) = \emptyset$, and that $U_{\mathcal{F}_0} = \emptyset$. Therefore, denoting $\mathcal{F}_i$ the sub-family $[C_1, C_2, ..., C_i]$ for all $0 \leq i \leq m + 1$:

- $\mathcal{F}_1 = [C_1]$ is a path-like family if C_1 contains at least one markable predicate for $U_{\mathcal{F}_0}$ and if C_1 maintains the path-like structure for $U_{\mathcal{F}_0}$,
- $\mathcal{F}_2 = [C_1, C_2]$ is a path-like family if $\mathcal{F}_1$ is path-like and a suitable clause $U_{\mathcal{F}_1}$ exists, and if C_2 satisfies the required properties with regard to $U_{\mathcal{F}_1}$,

- and so forth for $\mathcal{F}_i$, for i up to $m+1$, in which case $\mathcal{F}'$ is a path-like family.

The path-like structure of a family $\mathcal{F}'$ therefore depends on the existence of a sequence of appropriate clauses $U_{\mathcal{F}_i}$ for i from 1 up to $m+1$. The following proposition introduces a way to generate a clause $U_{\mathcal{F}'}$ when the family $\mathcal{F}' = [C_1, C_2, ..., C_m, C_{m+1}]$ is path-like, i.e. given the existence of a suitable clause $U_{\mathcal{F}}$ for $\mathcal{F} = [C_1, C_2, ..., C_m]$.

Proposition 1. *Given a path-like family $\mathcal{F}'$, consider a clause $U_{\mathcal{F}'}$ built as follows. Choose any markable predicate μ_{m+1} in C_{m+1} for $U_{\mathcal{F}}$. Let $\bar{C}_{m+1} = (C_{m+1} \setminus \{\mu_{m+1}\}) \cup \{\bar{\mu}_{m+1}\}$, and $U_{\mathcal{F}'} = U_{\mathcal{F}} \cup \bar{C}_{m+1}$. Such a clause $U_{\mathcal{F}'}$ always exists, its cover $X(U_{\mathcal{F}'})$ is nonempty and has no intersection with the reach of $\mathcal{F}'$ (i.e. $X(U_{\mathcal{F}'}) \cap \mathcal{R}(\mathcal{F}') = \emptyset$).*

Proof. The family $\mathcal{F}'$ being path-like, C_{m+1} maintains the path-like structure for $U_{\mathcal{F}}$, and we know there is at least one markable predicate $\mu_{m+1} \in C_{m+1}$ for $U_{\mathcal{F}}$. Therefore $X(U_{\mathcal{F}}) \cap X(\bar{C}_{m+1})$ is nonempty for $\bar{C}_{m+1}$ generated with any such μ_{m+1}.

Since $X(U_{\mathcal{F}}) \cap X(\bar{C}_{m+1}) = X(U_{\mathcal{F}} \cup \bar{C}_{m+1})$ and since $U_{\mathcal{F}'} = U_{\mathcal{F}} \cup \bar{C}_{m+1}$, we thus have that $X(U_{\mathcal{F}'}) = X(U_{\mathcal{F}}) \cap X(\bar{C}_{m+1})$. It follows that $X(U_{\mathcal{F}'})$ is nonempty.

We now prove by induction that $\mathcal{R}(\mathcal{F}') \cap X(U_{\mathcal{F}'}) = \emptyset$. Since we consider the reach of an empty family to be empty, the property is true for the initial family $\mathcal{F}_0 = \emptyset$.

Now suppose that $\mathcal{R}(\mathcal{F})$ and $X(U_{\mathcal{F}})$ have no intersection; i.e. $\mathcal{R}(\mathcal{F}) \cap X(U_{\mathcal{F}}) = \emptyset$. Recall that by construction, $U_{\mathcal{F}'} = U_{\mathcal{F}} \cup \bar{C}_{m+1}$.

On the one hand, since $\bar{C}_{m+1} \subseteq U_{\mathcal{F}'}$, it follows that $X(U_{\mathcal{F}'}) \subseteq X(\bar{C}_{m+1})$. Because $\mu_{m+1} \in C_{m+1}$ and $\bar{\mu}_{m+1} \in \bar{C}_{m+1}$, we know that the covers $X(\bar{C}_{m+1})$ and $X(C_{m+1})$ have no intersection. Therefore, it follows that the intersection $X(C_{m+1}) \cap X(U_{\mathcal{F}'})$ is empty.

On the other hand, since $U_{\mathcal{F}} \subsetneq U_{\mathcal{F}'}$, it follows that $X(U_{\mathcal{F}'}) \subsetneq X(U_{\mathcal{F}})$. According to the induction hypothesis, $\mathcal{R}(\mathcal{F}) \cap X(U_{\mathcal{F}}) = \emptyset$. Therefore the intersection $\mathcal{R}(\mathcal{F}) \cap X(U_{\mathcal{F}'})$ is also empty.

Finally, since $\mathcal{R}(\mathcal{F}') = \mathcal{R}(\mathcal{F}) \cup X(C_{m+1})$, it follows that $\mathcal{R}(\mathcal{F}') \cap X(U_{\mathcal{F}'}) = \emptyset$. □

Definition 8. Let us associate a unique **marked predicate** μ_i with each nogood clause C_i in a path-like family. Let $\mathcal{M} = [\mu_1, \mu_2, ..., \mu_m]$ be the set of marked predicates associated with $\mathcal{F} = [C_1, C_2, ..., C_m]$.

In Proposition 1, we define $U_{\mathcal{F}'} = U_{\mathcal{F}} \cup \bar{C}_{m+1}$, and recursively

$$U_{\mathcal{F}'} = \bigcup_{i=1}^{m+1} \bar{C}_i$$

where $\bar{C}_i = (C_i \setminus \{\mu_i\}) \cup \{\bar{\mu}_i\}$. It follows that if the family is path-like, then we can easily build a clause that covers a nonempty subset of $\mathcal{X}$ having no intersection with the reach of this family. We may then call `obstacle` with this clause in order to generate a new nogood clause S.

1.4 Updating the family

So far, the following assumptions on `obstacle` are required to hold at each iteration. Given any clause $U_{\mathcal{F}}$, denote by S the clause generated by `obstacle`$(U_{\mathcal{F}})$:

- S is a nogood clause,
- S satisfies $X(S) \cap X(U_{\mathcal{F}}) \neq \emptyset$.

In order to update the family $\mathcal{F}$ using the nogood clause S generated by `obstacle`$(U_{\mathcal{F}})$, we require the following additional assumptions to hold at each iteration:

- $\mathcal{F}$ is a path-like family,
- S maintains the path-like structure for $U_{\mathcal{F}}$.

Henceforth, we require no further assumptions on the procedure `obstacle`. Note that this is in contrast with previous work in [11] and [25] where the behavior of the `obstacle` procedure is much more precisely specified. They assume that

there is a 1-to-1 mapping from clauses to partial solutions, i.e. vectors whose components are not all instanciated. They also assume that the procedure `obstacle` is separable in two specific phases: a waxing phase and a waning phase, that can be described as follows, using our terminology. First, in the waxing phase, a clause U^+ is constructed from $U_{\mathcal{F}}$, and additional predicates are added to U^+ until it becomes nogood. Then, in the waning phase, a clause S is constructed from U^+, and predicates are removed from S one by one while maintaining the nogood property. The `obstacle` procedures satisfying such a specification are included in the set of `obstacle` procedures that we allow.

Example 4. Returning to our toy problem, using the `obstacle` procedure defined in example 2, we shall see that given any $U_{\mathcal{F}}$, the nogood clause $S = \texttt{obstacle}(U_{\mathcal{F}})$ always maintains the path-like structure for $U_{\mathcal{F}}$.

In this problem, the predicates in the clauses are bounds on l and h for $(l, h) \in \mathcal{X}$. Therefore, the cover of a clause is defined by the largest lower bound and the lowest upper bound on l and h: $\underline{l}$, $\bar{l}$, $\underline{h}$, $\bar{h}$ respectively. This is true in particular for the clause $U_{\mathcal{F}}$, and its cover is therefore always of the following type:

$$X(U_{\mathcal{F}}) = \left\{(l, h) \in \mathcal{X} : \underline{l} \leq l \leq \bar{l}, \underline{h} \leq h \leq \bar{h}\right\}.$$

Next, recall that the procedure `obstacle` is specified by arbitrarily selecting a point $(\tilde{l}, \tilde{h}) \in X(U_{\mathcal{F}})$. The procedure then infers a nogood clause S of the form $\{0 \leq l \leq \tilde{l}, 0 \leq h \leq \tilde{h}\}$ or $\{\tilde{l} \leq l \leq l_{\max}, \tilde{h} \leq h \leq h_{\max}\}$ when $(\tilde{l}, \tilde{h})$ is below or above Γ, respectively.

Finally, for each markable predicate $\gamma \in S$ for $U_{\mathcal{F}}$, it follows that for $\bar{S}_\gamma = (S \setminus \{\gamma\}) \cup \{\bar{\gamma}\}$ there is at least one point in both $X(\bar{S}_\gamma)$ and $X(U_{\mathcal{F}})$. Indeed, consider the following situations:

- If $\gamma = (\tilde{l} \leq l \leq l_{\max})$ then $(\tilde{l} - 1, \tilde{h}) \in X(\bar{S}_\gamma)$. Indeed, $X(\{\bar{\gamma}\}) \neq \emptyset$, hence $\tilde{l} \geq 1$. Also, since γ is not in $U_{\mathcal{F}}$, it follows that the largest lower bound on l in $U_{\mathcal{F}}$ is at most $\tilde{l} - 1$, thus $(\tilde{l} - 1, \tilde{h}) \in X(U_{\mathcal{F}})$. This case is illustrated in Figure 1.2.

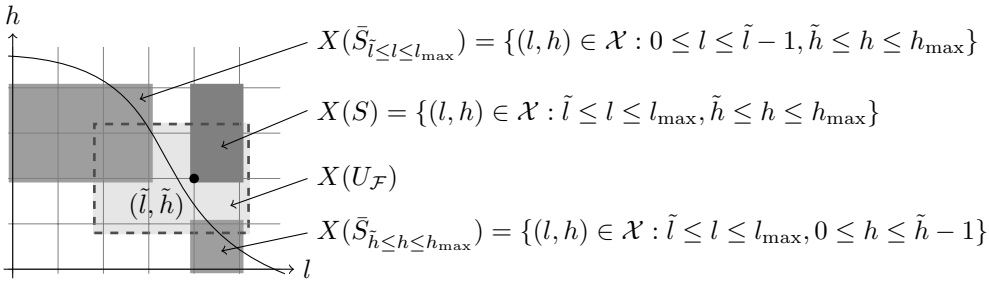


Figure 1.2: $S = \texttt{obstacle}(U_{\mathcal{F}})$ maintains the path-like structure for $U_{\mathcal{F}}$.

- If $\gamma = (0 \le l \le \tilde{l})$, $\gamma = (\tilde{h} \le h \le h_{\max})$, or $\gamma = (0 \le h \le \tilde{h})$, then using a similar argument, it follows that $(\tilde{l}+1, \tilde{h})$, $(\tilde{l}, \tilde{h}-1)$ and $(\tilde{l}, \tilde{h}+1)$ are in $X(\bar{S}_\gamma) \cap X(U_{\mathcal{F}})$, respectively. □

However, the path-like families and their update, as presented in this paper, are direct generalizations of the notions introduced by Chvátal [11]. There are two cases to consider: either S contains at least one markable predicate for $U_{\mathcal{F}}$, or it does not.

1.4.1 S contains at least one markable predicate for $U_{\mathcal{F}}$

In this case we can easily get a new path-like family $\mathcal{F}'$ from $\mathcal{F}$ and S by adding to $\mathcal{F}$ the clause S in position $m+1$; i.e. $\mathcal{F}' = [C_1, C_2, ..., C_m, C_{m+1}]$ where $C_{m+1} = S$.

Proposition 2. *Assuming that the procedure* `obstacle` *satisfies the assumptions presented at the beginning of this section, if S generated by* $\texttt{obstacle}(U_{\mathcal{F}})$ *contains at least one markable predicate for $U_{\mathcal{F}}$ and if $\mathcal{F} = [C_1, C_2, ..., C_m]$ is path-like, then the family $\mathcal{F}' = [C_1, C_2, ..., C_m, S]$ is path-like.*

Proof. Trivially, by defining $C_{m+1} = S$, it follows that C_{m+1} contains at least one markable predicate for $U_{\mathcal{F}}$. Since S was generated by $\texttt{obstacle}(U_{\mathcal{F}})$, it follows that C_{m+1} maintains the path-like structure for $U_{\mathcal{F}}$. Hence, $\mathcal{F}'$ is path-like by definition 7. □

In order to generate the clause $U_{\mathcal{F}'}$ to be used with `obstacle` at the next iteration, we proceed as in Proposition 1.

Example 5. Referring to our toy problem, this case can be illustrated as follows. Suppose we have

$$\mathcal{F} = \begin{bmatrix} \{\underline{0 \leq l \leq 2}, 0 \leq h \leq 3\} \\ \{0 \leq l \leq 4, \underline{0 \leq h \leq 1}\} \end{bmatrix}$$

where the underlined relations correspond to the marked predicates in $\mathcal{M}$. Accordingly, $U_{\mathcal{F}} = \{3 \leq l \leq 5, 0 \leq h \leq 3, 0 \leq l \leq 4, 2 \leq h \leq 4\}$.

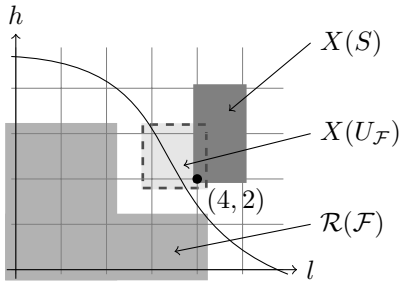


Figure 1.3: Search state at this iteration.

To generate S, suppose that the point $(4, 2)$ is selected arbitrarily by `obstacle` in $X(U_{\mathcal{F}}) = \{(l, h) \in \mathbb{Z}^2 : 3 \leq l \leq 4, 2 \leq h \leq 3\}$. Note that $(4, 2)$ lies above Γ. Thus $S = \{4 \leq l \leq 5, 2 \leq h \leq 4\}$, as illustrated in Figure 1.3.

Since the predicate $(4 \leq l \leq 5)$ in S is markable for $U_{\mathcal{F}}$, the family $\mathcal{F}'$ is generated by adding to $\mathcal{F}$ the clause S in position 3. The predicate $\mu_S = (4 \leq l \leq 5)$ is selected as the marked predicate for the last clause, and it follows that

$$\mathcal{F}' = \begin{bmatrix} \{\underline{0 \leq l \leq 2}, 0 \leq h \leq 3\} \\ \{0 \leq l \leq 4, \underline{0 \leq h \leq 1}\} \\ \{\underline{4 \leq l \leq 5}, 2 \leq h \leq 4\} \end{bmatrix}$$

and $U_{\mathcal{F}'} = \{3 \leq l \leq 5, 0 \leq h \leq 3, 0 \leq l \leq 4, 2 \leq h \leq 4, 0 \leq l \leq 3, 2 \leq h \leq 4\}$, as illustrated in Figure 1.4. □

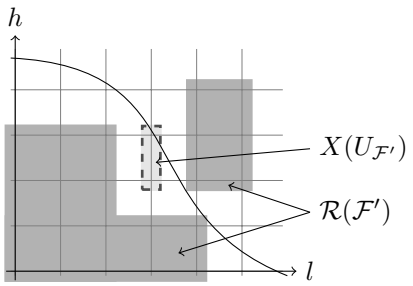


Figure 1.4: Search state after this iteration.

1.4.2 S contains no markable predicate for $U_{\mathcal{F}}$

In this case, adding the clause S at the end of $\mathcal{F}$ does not infer a path-like family. However, it is possible to deduce a new path-like family of nogood clauses using S and $\mathcal{F}$. But first, we have to introduce the notion of **clause resolution**.

Definition 9. Let A and B be two clauses such that there is one and only one predicate γ such that $\gamma \in A$ and $\bar{\gamma} \in B$. The **resolvent** of A and B is defined as the clause:

$$A \nabla B = (A \setminus \{\gamma\}) \cup (B \setminus \{\bar{\gamma}\}).$$

The following result shows that the resolution operation to define the resolvent of two nogood clauses A and B preserves the nogood property of the resolvent $A \nabla B$.

Proposition 3. *If A and B are two nogood clauses having a resolvent $A \nabla B$, then $A \nabla B$ is also a nogood clause.*

Proof. Denote $A = A' \cup \{\gamma\}$ and $B = B' \cup \{\bar{\gamma}\}$. Hence $A \nabla B = A' \cup B'$.

Since γ and $\bar{\gamma}$ are complement of each other, it follows by definition that any solution in $\mathcal{X}$ satisfies either γ or $\bar{\gamma}$. In particular, this is true for any solution $x \in X(A' \cup B')$.

Therefore, if x satisfies γ, then since it also satisfies A', it also satisfies A. Otherwise, x satisfies both $\bar{\gamma}$ and B', and hence B. It follows that any solution $x \in X(A' \cup B')$ satisfies A or B. The assumption that A and B are nogood clauses implies that $A \nabla B = A' \cup B'$ is also nogood. □

This result is a generalization of the clause resolution mechanism for integer programming problems presented in [25], which itself is a generalization of clause resolution as presented in [11] in the context of 0-1 problems. It allows to introduce the following procedure in algorithm 2, which generates recursively a new nogood clause R using the nogood clause S generated by `obstacle`$(U_{\mathcal{F}})$ and some nogood clauses in $\mathcal{F}$.

Algorithm 2 - Procedure `ResolventGeneration`$(S, \mathcal{F}, \mathcal{M})$:

Step 0. Let $R = S$ and $i = m$.

Step 1. $\mu_i \in \mathcal{M}$ is the marked predicate associated with the nogood $C_i \in \mathcal{F}$. If its complement $\bar{\mu}_i$ is in R, replace R by $R \nabla C_i$.

Step 2. Decrement i. If $i = 0$, then return R, else go to Step 1.

Proposition 4. *Let S be the nogood clause generated by* `obstacle`$(U_{\mathcal{F}})$*. If S contains no markable predicate for $U_{\mathcal{F}}$, then the clause R generated by* `ResolventGeneration`$(S, \mathcal{F}, \mathcal{M})$ *is nogood, and R contains no markable predicate for $U_{\mathcal{F}}$ either.*

Proof. We prove this proposition by induction. At the beginning of the `ResolventGeneration` procedure when $i = m$, $R = S$. S is a nogood clause and contains no markable predicate for $U_{\mathcal{F}}$, hence the proposition is true for $i = m$.

Suppose that at the beginning of iteration i of `ResolventGeneration`, R is a nogood clause and R contains no markable predicate for $U_{\mathcal{F}}$. If $\bar{\mu}_i$ is not in R, then R is not modified during this iteration. Now suppose that $\bar{\mu}_i \in R$. We first show that $\mu_i \in C_i$ is the unique element of C_i allowing R and C_i to have a resolvent.

For contradiction, suppose that there is another predicate $\gamma \neq \mu_i$ such that $\gamma \in C_i$ and $\bar{\gamma} \in R$. On the one hand, since R contains no markable predicate for $U_{\mathcal{F}}$, all predicates in R are either in $U_{\mathcal{F}}$ or are satisfied for all elements in $\mathcal{X}$. We know $X(C_i) \neq \emptyset$ and we have $\gamma \in C_i$, it follows that $\bar{\gamma} \in U_{\mathcal{F}}$. On the other hand, we have $\bar{C}_i = (C_i \setminus \{\mu_i\}) \cup \{\bar{\mu}_i\}$. Thus since $\gamma \in C_i$ and $\gamma \neq \mu_i$, we have $\gamma \in \bar{C}_i$. Therefore it follows that $\gamma \in U_{\mathcal{F}} = \cup_{j=1}^{m} \bar{C}_j$. But then $\gamma \in U_{\mathcal{F}}$ and $\bar{\gamma} \in U_{\mathcal{F}}$, a contradiction

because $X(U_{\mathcal{F}}) \neq \emptyset$. Therefore the resolvent $R \nabla C_i = (R \setminus \{\bar{\mu}_i\}) \cup (C_i \setminus \{\mu_i\})$ exists and is a nogood clause by Proposition 3.

By the induction hypothesis, $(R \setminus \{\bar{\mu}_i\})$ contains no markable predicate for $U_{\mathcal{F}}$. Also, because $U_{\mathcal{F}} = \cup_{j=1}^{m} \bar{C}_j$ where $\bar{C}_j = (C_j \setminus \{\mu_j\}) \cup \{\bar{\mu}_j\}$, we have that $(C_i \setminus \{\mu_i\}) \subsetneq \bar{C}_i \subseteq U_{\mathcal{F}}$. Therefore it follows that $R \nabla C_i$ contains no markable predicate for $U_{\mathcal{F}}$. □

We now show that if R does not cover the entire search space, there exists an index k, $1 \leq k \leq m$, such that a new path-like family can be generated by removing all the clauses C_i, $k \leq i \leq m$, from $\mathcal{F}$, and by adding the nogood clause R.

To ease the presentation, $\mathcal{F}_i = [C_1, C_2, ..., C_i]$ denotes the sub-families for $i = 1, ..., m$, and $\mathcal{F}_0 = \emptyset$. Because $\mathcal{F}$ is path-like, $\mathcal{F}_i$ is also necessarily path-like.

Proposition 5. *Let R be the nogood clause generated by* `ResolventGeneration`$(S, \mathcal{F}, \mathcal{M})$. *Let k be the smallest index such that R contains no markable predicate for $U_{\mathcal{F}_k}$. If $k = 0$, then the search is completed, otherwise the family $\mathcal{F}' = [C_1, C_2, ..., C_{k-1}, R]$ is path-like.*

Proof. If $k = 0$, R contains no markable predicate for $U_{\mathcal{F}_0} = \emptyset$, this implies $X(R) = \mathcal{X}$ and since R is a nogood clause, it follows that the search space is completely explored.

Otherwise, we have $1 \leq k$, as well as $k \leq m$ since we know that R contains no markable predicate for $U_{\mathcal{F}} = U_{\mathcal{F}_m}$. Furthermore, since k is the smallest index such that R contains no markable predicate for $U_{\mathcal{F}_k}$, it also follows that R contains at least one markable predicate for $U_{\mathcal{F}_{k-1}}$. To complete the proof, we have to show that R maintains the path-like structure for $U_{\mathcal{F}_{k-1}}$, since the family $\mathcal{F}_{k-1}$ is path-like.

Since the clause R contains no markable predicate for $U_{\mathcal{F}_k} = U_{\mathcal{F}_{k-1}} \cup \bar{C}_k$, then it follows that all markable predicates in R for $U_{\mathcal{F}_{k-1}}$ (there is at least one) are in $\bar{C}_k \setminus U_{\mathcal{F}_{k-1}}$. Recall that $\bar{\mu}_k \notin U_{\mathcal{F}_{k-1}}$, and by construction $\bar{\mu}_k \notin R$, thus all markable predicates in R for $U_{\mathcal{F}_{k-1}}$ are also in $(\bar{C}_k \setminus \{\bar{\mu}_k\}) \setminus U_{\mathcal{F}_{k-1}}$, and thus also in $C_k \setminus U_{\mathcal{F}_{k-1}}$. Since the family $\mathcal{F}_k$ is path-like, C_k is a clause which maintains the

path-like structure for $U_{\mathcal{F}_{k-1}}$, hence R also maintains the path-like structure for $U_{\mathcal{F}_{k-1}}$. □

Example 6. To illustrate this case using our toy problem, suppose that we are at the beginning of an iteration where $\bar{z} = -6$ and

$$\mathcal{F} = \begin{bmatrix} \{\underline{0 \le l \le 2}, 0 \le h \le 3\} \\ \{0 \le l \le 4, \underline{0 \le h \le 1}\} \\ \{\underline{4 \le l \le 5}, 2 \le h \le 4\} \end{bmatrix} \begin{matrix} : C_1 \\ : C_2 \\ : C_3, \end{matrix}$$

the underlined relations corresponding to the marked predicates in $\mathcal{M}$. Accordingly, $U_{\mathcal{F}} = \{3 \le l \le 5, 0 \le h \le 3, 0 \le l \le 4, 2 \le h \le 4, 0 \le l \le 3, 2 \le h \le 4\}$.

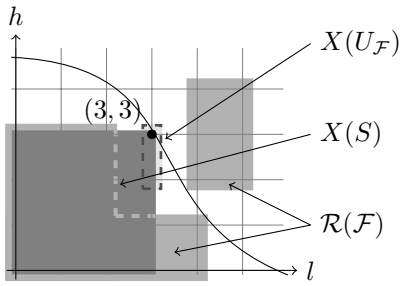


Figure 1.5: Search state at this iteration.

To generate S, suppose that the point $(3, 3)$ is selected arbitrarily by `obstacle` in $X(U_{\mathcal{F}}) = \{(l, h) \in \mathbb{Z}^2 : 3 \le l \le 3, 2 \le h \le 3\}$. Note that $(3, 3)$ lies below Γ. Thus $S = \{0 \le l \le 3, 0 \le h \le 3\}$ as illustrated in Figure 1.5. Furthermore, since $f(3, 3) < -6$, the `obstacle` procedure updates $\bar{z} = -9$ and $r^* = (3, 3)$.

Since $S \subset U_{\mathcal{F}}$, it follows that S contains no markable predicates for $U_{\mathcal{F}}$. Hence we first generate R from the resolvents of S and the clauses in $\mathcal{F} = [C_1, C_2, C_3]$. Initiate the process with $R = S$ and $i = 3$.

$i = 3$**:** $\mu_3 = (4 \le l \le 5)$ implies that $\bar{\mu}_3 = (0 \le l \le 3)$, and this predicate is in R.
Replace R with $R \nabla C_3 = (R \setminus \{\bar{\mu}_3\}) \cup (C_3 \setminus \{\mu_3\}) = \{0 \le h \le 3, 2 \le h \le 4\}$.

$i = 2$**:** $\mu_2 = (0 \leq h \leq 1)$ implies that $\bar{\mu}_2 = (2 \leq h \leq 4)$, and this predicate is in R. Replace R with $R \nabla C_2 = \{0 \leq h \leq 3, 0 \leq l \leq 4\}$.

$i = 1$**:** $\mu_1 = (0 \leq l \leq 2)$ implies that $\bar{\mu}_1 = (3 \leq l \leq 5)$, and this predicate is not in R.

The resulting nogood clause is $R = \{0 \leq h \leq 3, 0 \leq l \leq 4\}$.

Now determine the rank k such that R contains no markable predicates for $U_{\mathcal{F}_k}$. Initiate the process with $k = 0$ and $U_{\mathcal{F}_0} = \emptyset$.

$k = 0$**:** $(0 \leq h \leq 3) \in R$ is a markable predicate for $U_{\mathcal{F}_0} = \emptyset$.

$k = 1$**:** $(0 \leq l \leq 4) \in R$ is a markable predicate for $U_{\mathcal{F}_1} = U_{\mathcal{F}_0} \cup \bar{C}_1 = \{3 \leq l \leq 5, 0 \leq h \leq 3\}$.

$k = 2$**:** R contains no markable predicates for $U_{\mathcal{F}_2} = U_{\mathcal{F}_1} \cup \bar{C}_2 = \{3 \leq l \leq 5, 0 \leq h \leq 3, 0 \leq l \leq 4, 2 \leq h \leq 4\}$.

Thus $\mathcal{F}' = [C_1, R]$, and $\mu_R = (0 \leq l \leq 4)$ is selected as the marked predicate for the clause R. It follows that

$$\mathcal{F}' = \begin{bmatrix} \{\underline{0 \leq l \leq 2}, 0 \leq h \leq 3\} \\ \{\underline{0 \leq l \leq 4}, 0 \leq h \leq 3\} \end{bmatrix} \begin{matrix} : C_1 \\ : R \end{matrix},$$

and $U_{\mathcal{F}'} = \{3 \leq l \leq 5, 0 \leq h \leq 3, 5 \leq l \leq 5, 0 \leq h \leq 3\}$, as illustrated in Figure 1.6. □

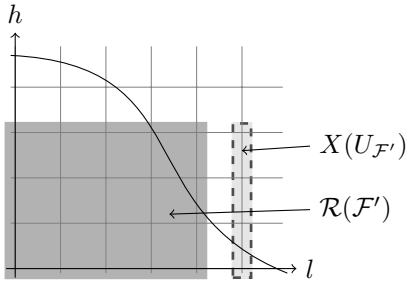


Figure 1.6: Search state after this iteration.

1.5 Complete algorithm and convergence

The Resolution Search approach can now be fully summarized in algorithm 3.

Algorithm 3 - Procedure `ResolutionSearch`:

Step 0 - Initialization.
Let $\mathcal{F} = \emptyset$, $\mathcal{M} = \emptyset$, $U_{\mathcal{F}} = \emptyset$ and $m = 0$.

Step 1 - Exploration.
Let S be the clause generated by `obstacle`$(U_{\mathcal{F}})$.

Step 2 - Construction.
Perform either Step 2.1 or Step 2.2.

Step 2.1 - Case S contains at least one markable predicate for $U_{\mathcal{F}}$.
Select $\mu_S \in S$, a markable predicate for $U_{\mathcal{F}}$. Let $m' = m + 1$, $\mathcal{F}' = [C_1, ..., C_m, S]$ and $\mathcal{M}' = [\mu_1, ..., \mu_m, \mu_S]$. Let $\bar{S} = (S \setminus \{\mu_S\}) \cup \{\bar{\mu}_S\}$ and $U_{\mathcal{F}'} = U_{\mathcal{F}} \cup \bar{S}$. Go to Step 3.

Step 2.2 - Case S contains no markable predicate for $U_{\mathcal{F}}$.

Step 2.2.1 - Generate R.
Let R be the nogood clause generated by `ResolventGeneration`$(S, \mathcal{F}, \mathcal{M})$. Let k be the smallest index such that R contains no markable predicate for $U_{\mathcal{F}_k}$.

Step 2.2.2 - Generate $\mathcal{F}'$.
If $k = 0$, return the best known solution, the search is completed. Otherwise, select $\mu_R \in R$, a markable predicate for $U_{\mathcal{F}_{k-1}}$. Let $m' = k$, $\mathcal{F}' = [C_1, ..., C_{k-1}, R]$ and $\mathcal{M}' = [\mu_1, ..., \mu_{k-1}, \mu_R]$. Let $\bar{R} = (R \setminus \{\mu_R\}) \cup \{\bar{\mu}_R\}$ and $U_{\mathcal{F}'} = U_{\mathcal{F}_{k-1}} \cup \bar{R}$.

Step 3 - Update.
Replace $\mathcal{F}$ by $\mathcal{F}'$, $\mathcal{M}$ by $\mathcal{M}'$, $U_{\mathcal{F}}$ by $U_{\mathcal{F}'}$, and m by m'. Return to Step 1.

To prove the convergence of the Resolution Search procedure, we cannot rely on the strict increase of the reach $\mathcal{R}(\mathcal{F})$ at each iteration. Note that Chvátal also observes this for a 0-1 programming problem in [11]. However, the convergence proof of the Resolution Search procedure relies on a subset of the reach $\mathcal{R}(\mathcal{F})$ that is strictly increasing at each iteration.

Definition 10. Given $\mathcal{F} = [C_1, C_2, ..., C_m]$ a path-like family and the associated set of marked predicates $\mathcal{M} = [\mu_1, \mu_2, ..., \mu_m]$, the **restricted reach** $\check{\mathcal{R}}(\mathcal{F})$ of the family $\mathcal{F}$ is defined as follows:

$$\check{\mathcal{R}}(\mathcal{F}) = \bigcup_{i=1}^{m} X(\cup_{j=1}^{i-1} \bar{C}_j \cup C_i) = \bigcup_{i=1}^{m} X(U_{\mathcal{F}_{i-1}} \cup C_i).$$

Note that the notion of restricted reach is equivalent to the σ strength function of Chvátal [11], since $|\check{\mathcal{R}}(\mathcal{F})| = \sigma(\mathcal{F})$ for 0-1 problems.

Clearly, the restricted reach of any family is a subset of its reach. We consider two different lemmas to show that the restricted reach strictly increases at each iteration according to the way of generating the new family $\mathcal{F}'$.

Lemma 1. *Let S be the nogood clause generated by* `obstacle`$(U_{\mathcal{F}})$*. If S contains at least one markable predicate for $U_{\mathcal{F}}$, the family $\mathcal{F}'$ generated in Step 2.1 of the Resolution Search procedure satisfies $\check{\mathcal{R}}(\mathcal{F}) \subsetneq \check{\mathcal{R}}(\mathcal{F}')$.*

Proof. By definition of the restricted reach and of the clause $U_{\mathcal{F}}$:

$$\check{\mathcal{R}}(\mathcal{F}') = \check{\mathcal{R}}(\mathcal{F}) \cup X(\cup_{j=1}^{m} \bar{C}_j \cup S) = \check{\mathcal{R}}(\mathcal{F}) \cup X(U_{\mathcal{F}} \cup S).$$

The proof is completed if we can show that $X(U_{\mathcal{F}} \cup S)$ is nonempty and not in $\check{\mathcal{R}}(\mathcal{F})$. By definitions of S and `obstacle`, $X(U_{\mathcal{F}} \cup S) = X(U_{\mathcal{F}}) \cap X(S) \neq \emptyset$. Also, since $X(U_{\mathcal{F}} \cup S) \subset X(U_{\mathcal{F}})$ and since $X(U_{\mathcal{F}}) \cap \mathcal{R}(\mathcal{F}) = \emptyset$, it follows that $X(U_{\mathcal{F}} \cup S) \cap \mathcal{R}(\mathcal{F}) = \emptyset$. Hence, since $\check{\mathcal{R}}(\mathcal{F}) \subseteq \mathcal{R}(\mathcal{F})$, it follows that $X(U_{\mathcal{F}} \cup S) \cap \check{\mathcal{R}}(\mathcal{F}) = \emptyset$. □

Lemma 2. *Let S be the nogood clause generated by* `obstacle`$(U_{\mathcal{F}})$*. If S contains no markable predicate for $U_{\mathcal{F}}$, the family $\mathcal{F}'$ generated in Step 2.2 of the Resolution Search procedure satisfies $\check{\mathcal{R}}(\mathcal{F}) \subsetneq \check{\mathcal{R}}(\mathcal{F}')$.*

Proof. In this case $\mathcal{F}' = [C_1, C_2, ..., C_{k-1}, R]$ is built by adding R to $\mathcal{F}_{k-1}$. By definitions of the restricted reach and of the clause $U_{\mathcal{F}_{k-1}}$:

$$\check{\mathcal{R}}(\mathcal{F}') = \check{\mathcal{R}}(\mathcal{F}_{k-1}) \cup X(\cup_{j=1}^{k-1} \bar{C}_j \cup R) = \check{\mathcal{R}}(\mathcal{F}_{k-1}) \cup X(U_{\mathcal{F}_{k-1}} \cup R).$$

To show that $\check{\mathcal{R}}(\mathcal{F}) \subset \check{\mathcal{R}}(\mathcal{F}')$, consider any solution $x \in \check{\mathcal{R}}(\mathcal{F})$. Then $x \in X(U_{\mathcal{F}_{i-1}} \cup C_i)$ for at least one index i, $1 \leq i \leq m$.

- If $i < k$, then $x \in \check{\mathcal{R}}(\mathcal{F}_{k-1})$, and thus $x \in \check{\mathcal{R}}(\mathcal{F}')$.
- If $i = k$, then $x \in X(U_{\mathcal{F}_{k-1}} \cup C_k) \subsetneq X(U_{\mathcal{F}_{k-1}} \cup (C_k \setminus \{\mu_k\}))$. However, R contains no markable predicate for $U_{\mathcal{F}_k}$, and hence $X(U_{\mathcal{F}_k}) \subseteq X(R)$. Since neither μ_k nor $\bar{\mu}_k$ can belong to R, and since $U_{\mathcal{F}_k} = U_{\mathcal{F}_{k-1}} \cup \bar{C}_k$, it follows that $X(U_{\mathcal{F}_{k-1}} \cup (\bar{C}_k \setminus \{\bar{\mu}_k\})) \subseteq X(R)$. Therefore $x \in X(U_{\mathcal{F}_{k-1}} \cup R)$, and thus $x \in \check{\mathcal{R}}(\mathcal{F}')$.
- If $i > k$, then $x \in X(U_{\mathcal{F}_{i-1}} \cup C_i)$. Since $i > k$, we have $X(U_{\mathcal{F}_{i-1}} \cup C_i) \subsetneq X(U_{\mathcal{F}_k})$. As seen when $i = k$, we also have $X(U_{\mathcal{F}_k}) \subsetneq X(U_{\mathcal{F}_{k-1}} \cup R)$. Therefore $x \in X(U_{\mathcal{F}_{k-1}} \cup R)$, and thus $x \in \check{\mathcal{R}}(\mathcal{F}')$.

Next we show the inclusion to be strict.

The clause R contains no markable predicate for $U_{\mathcal{F}_k}$, and since $U_{\mathcal{F}_{k-1}} \subsetneq U_{\mathcal{F}_k} \subseteq U_{\mathcal{F}}$, R contains no markable predicate for $U_{\mathcal{F}}$ either, hence $X(U_{\mathcal{F}})$ is a subset of both $X(R)$ and $X(U_{\mathcal{F}_{k-1}})$. Therefore $X(U_{\mathcal{F}}) \subseteq X(U_{\mathcal{F}_{k-1}}) \cap X(R)$, or equivalently $X(U_{\mathcal{F}}) \subseteq X(U_{\mathcal{F}_{k-1}} \cup R)$. Then $X(U_{\mathcal{F}}) \subseteq \check{\mathcal{R}}(\mathcal{F}')$. However, $\check{\mathcal{R}}(\mathcal{F}) \subseteq \mathcal{R}(\mathcal{F})$ and $X(U_{\mathcal{F}}) \cap \mathcal{R}(\mathcal{F}) = \emptyset$. Thus since $X(U_{\mathcal{F}})$ is nonempty, it follows that there exists at least one solution in the restricted reach of $\mathcal{F}'$ that does not belong to the restricted reach of $\mathcal{F}$. □

Theorem 1. *The Resolution Search procedure completely explores $\mathcal{X}$ in at most $|\mathcal{X}|$ iterations.*

Proof. According to the previous lemmas, $\check{\mathcal{R}}(\mathcal{F}) \subsetneq \check{\mathcal{R}}(\mathcal{F}')$ at every iteration. Since the restricted reach of a family is included in its reach, it follows that at each iteration, a subset of the reach of $\mathcal{F}$ increases strictly. In the worst case, it increases by exactly one solution at each iteration, and thus at most $|\mathcal{X}|$ iterations are needed for the search to complete. □

Having shown that the Resolution Search procedure converges when $\mathcal{X}$ is finite, let us conclude first by addressing a few points originally brought up by Chvátal in [11] concerning the re-use of nogood clauses which are discarded from the path-like family during the search, then by arguing which kind of problems might benefit from a Generalized Resolution Search approach.

1.6 Concluding remarks

1.6.1 Recycling nogood clauses

When the new family $\mathcal{F}'$ is generated via Step 2.2 of the Resolution Search procedure, the nogood clauses $C_k, ..., C_m$, and S are discarded and we have $\mathcal{F}' = [C_1, C_2, ..., C_{k-1}, R]$. However, some of these clauses may have a cover intersecting $X(U_{\mathcal{F}'})$; i.e. $X(U_{\mathcal{F}'}) \cap X(C_i) \neq \emptyset$ for some $i = k, ..., m$, or $X(U_{\mathcal{F}'}) \cap X(S) \neq \emptyset$. If one of these nogood clauses also maintains the path-like structure for $U_{\mathcal{F}'}$, then it could be used in place of the nogood clause to be generated by `obstacle`$(U_{\mathcal{F}})$ at the next iteration. The computational effort is then reduced at the next iteration, and this may diminish the overall effort required to solve the problem.

1.6.2 Example of a possible application

Obviously, one contribution of this paper is to extend Resolution Search beyond 0-1 problems to all discrete problems, MIP included. However, and perhaps more interestingly, another contribution is to specify Resolution Search for a much larger class of `obstacle` procedures than previously allowed, even for 0-1 problems. Consider for example the Generalized Assignment Problem (GAP), which can be modelled by the following 0-1 program:

$$\min \qquad \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij}$$

$$\text{subject to} \qquad \sum_{j=1}^{n} a_{ij}x_{ij} \leq b_i, \qquad i = 1, \ldots, m. \tag{1.1}$$

$$\sum_{i=1}^{m} x_{ij} = 1, \qquad j = 1, \ldots, n. \tag{1.2}$$

$$0 \leq x_{ij} \leq 1, \qquad i = 1, \ldots, m; j = 1, \ldots, n. \tag{1.3}$$

$$x_{ij} \text{ integer}, \quad i = 1, \ldots, m; j = 1, \ldots, n. \tag{1.4}$$

The GAP consists in assigning n jobs, indexed by j, to m agents, indexed by i. Assuming the costs c are all nonpositive, the constraints (1.2) can be modelled as Special Ordered Sets of type I (SOS1) [1] [4] [27], which are defined to be independent and ordered sets of variables in which at most one member may be non-zero in a feasible solution. Modern Branch-and-Bound solvers use special branching strategies to take advantage of SOS1s: branching consists in dividing a SOS1 into two sub-sequences and fixing the variables in each to zero. When applied to solving the GAP, such strategies consistently yield smaller search trees than the usual fractional variable branching strategy.

We can apply this idea of forbidding several assignments at once within a Generalized Resolution Search approach. In Resolution Search as proposed in [11], such a strategy is not possible because each 0-1 variable must be considered separately. Let us use predicates of the following type:

- given a job $1 \leq j \leq n$ and a subset of agents $I \subsetneq \{1, \ldots, m\}$, the predicate $\gamma(j, I)$ is satisfied by a solution $\bar{x}$ if and only if $\bar{x}_{ij} = 0$ for all $i \in I$,
- consequently the predicate $\gamma(j, \{1, \ldots, m\} \setminus I)$ is the complement predicate of $\gamma(j, I)$, and is therefore of the same type.

Here is an overview of a suitable `obstacle` procedure for the GAP:

1. Find a solution $\bar{x} \in X(U_{\mathcal{F}})$. Find a subset of solutions $X(C)$ which has $\bar{x}$ as a minimum. Update the upper bound $\bar{z}$ if necessary.
2. Generate and return a clause S such that:

- S is composed only of predicates of the type $\gamma(j, I)$,
- $\bar{x} \in X(S) \subseteq X(C)$,
- and S maintains the path-like structure for $U_{\mathcal{F}}$.

The search for a solution $\bar{x}$ can be done heuristically, for example by applying a greedy algorithm in $X(U_{\mathcal{F}})$. The identification of a subset $X(C)$ can be done by exploiting the structure of the problem, for example by using dual information as suggested by Chvátal [11].

Generating S using only predicates of the type $\gamma(j, I)$ instead of any others does not change the way `obstacle` searches the subset of solutions $X(U_{\mathcal{F}})$. However, the overall performance of the search also depends on the characteristics of $X(U_{\mathcal{F}})$ at each call to `obstacle`: intuitively, the larger and better (in terms of solution quality) it is, the easier it is to generate clauses S which will substantially increase the restricted reach of the path-like family. A consequence of our choice of predicates is that it is easier to obtain clauses $U_{\mathcal{F}}$ at each iteration which yield "consistently good" subsets $X(U_{\mathcal{F}})$ for `obstacle` to search in.

1.6.3 Acknowledgements

The authors wish to thank the anonymous referees for their helpful comments and suggestions.

1.7 Appendix

In this appendix, we continue the application of Resolution Search to our toy problem after the iteration in Example 6, referred to as iteration n. Recall that the selection of the marked predicates is arbitrary, and nogood clauses are not recycled.

The notational conventions, the symbols and the colors are used as previously. For instance, in the figures, the cover of $U_{\mathcal{F}}$ is displayed in light gray, the reach of $\mathcal{F}$ is in medium gray, and the cover of S is in dark gray. When displaying $\mathcal{F}$, the clauses are stacked from top to bottom, and the marked predicates in each clause

(i.e. the elements in $\mathcal{M}$) are underlined.

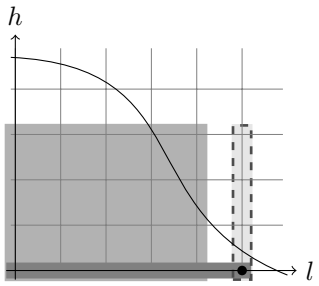
Recall that at the end of Example 6, i.e. at the end of iteration n, we had $\bar{z} = 9$ and

$$\mathcal{F}' = \begin{bmatrix} \{\underline{0 \leq l \leq 2}, 0 \leq h \leq 3\} \\ \{\underline{0 \leq l \leq 4}, 0 \leq h \leq 3\} \end{bmatrix},$$

and thus $U_{\mathcal{F}} = \{3 \leq l \leq 5, 0 \leq h \leq 3, 5 \leq l \leq 5, 0 \leq h \leq 3\}$.

Iteration $n + 1$.

To generate S, suppose that the point $(5, 0)$ is selected arbitrarily by `obstacle` in $X(U_{\mathcal{F}}) = \{(l, h) \in \mathbb{Z}^2 : 5 \leq l \leq 5, 0 \leq h \leq 3\}$. This point lies below Γ, and thus `obstacle` generates $S = \{0 \leq l \leq 5, 0 \leq h \leq 0\}$.



Search state at iteration $n + 1$.

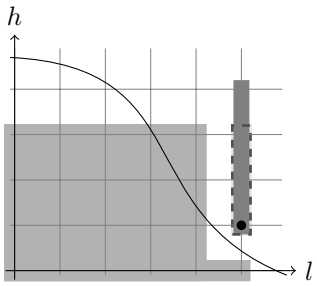
Since S contains a markable predicate for $U_{\mathcal{F}}$, the family $\mathcal{F}'$ is generated by adding S to $\mathcal{F}$. The predicate $\mu_S = (0 \leq h \leq 0)$ is selected as the marked predicate for that clause. It follows that

$$\mathcal{F}' = \begin{bmatrix} \{\underline{0 \leq l \leq 2}, 0 \leq h \leq 3\} \\ \{\underline{0 \leq l \leq 4}, 0 \leq h \leq 3\} \\ \{0 \leq l \leq 5, \underline{0 \leq h \leq 0}\} \end{bmatrix}$$

and $U_{\mathcal{F}'} = \{3 \leq l \leq 5, 0 \leq h \leq 3, 5 \leq l \leq 5, 0 \leq h \leq 3, 0 \leq l \leq 5, 1 \leq h \leq 4\}$.

Iteration $n+2$.

To generate S, suppose that the point $(5,1)$ is selected arbitrarily by `obstacle` in $X(U_{\mathcal{F}}) = \{(l,h) \in \mathbb{Z}^2 : 5 \leq l \leq 5, 1 \leq h \leq 3\}$. This point lies above Γ, and thus `obstacle` generates $S = \{5 \leq l \leq 5, 1 \leq h \leq 4\}$.



Search state at iteration $n+2$.

Since S contains no markable predicate for $U_{\mathcal{F}}$, we first generate R from the resolvents of S and the clauses in $\mathcal{F} = [C_1, C_2, C_3]$. Initiate the process with $R = S$ and $i = 3$.

$i = 3$**:** $\mu_3 = (0 \leq h \leq 0)$ implies that $\bar{\mu}_3 = (1 \leq h \leq 4)$, and this predicate is in R.
Replace R with $R \nabla C_3 = (R \setminus \{\bar{\mu}_3\}) \cup (C_3 \setminus \{\mu_3\}) = \{5 \leq l \leq 5, 0 \leq l \leq 5\}$.

$i = 2$**:** $\mu_2 = (0 \leq l \leq 4)$ implies that $\bar{\mu}_1 = (5 \leq l \leq 5)$, and this predicate is in R.
Replace R with $R \nabla C_2 = (R \setminus \{\bar{\mu}_2\}) \cup (C_2 \setminus \{\mu_2\}) = \{0 \leq l \leq 5, 0 \leq h \leq 3\}$.

$i = 1$**:** $\mu_1 = (0 \leq l \leq 2)$ implies that $\bar{\mu}_1 = (3 \leq l \leq 5)$, and this predicate is not in R.

The resulting nogood clause is $R = \{0 \leq l \leq 5, 0 \leq h \leq 3\}$.

Now determine the rank k such that R contains no markable predicate for $U_{\mathcal{F}_k}$. Initiate the process with $k = 0$ and $U_{\mathcal{F}_0} = \emptyset$.

$k = 0$**:** $(0 \leq h \leq 3) \in R$ is a markable predicate for $U_{\mathcal{F}_0} = \emptyset$.

$k = 1$**:** R contains no markable predicates for $U_{\mathcal{F}_1} = U_{\mathcal{F}_0} \cup \bar{C}_1 = \{3 \leq l \leq 5, 0 \leq h \leq 3\}$.

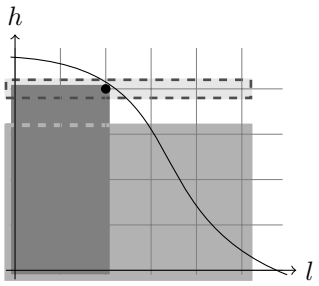
Thus $\mathcal{F}' = [R]$, and $\mu_R = (0 \leq h \leq 3)$ is selected as the marked predicate for the clause R. It follows that

$$\mathcal{F}' = \left[\ \{0 \leq l \leq 5, \underline{0 \leq h \leq 3}\} \ \right]$$

and $U_{\mathcal{F}'} = \{0 \leq l \leq 5, 4 \leq h \leq 4\}$.

Iteration $n + 3$**.**

To generate S, suppose the point $(2, 4)$ is selected arbitrarily by `obstacle` in $X(U_{\mathcal{F}}) = \{(l, h) \in \mathbb{Z}^2 : 0 \leq l \leq 5, 4 \leq h \leq 4\}$. This point lies below Γ, and thus `obstacle` generates $S = \{0 \leq l \leq 2, 0 \leq h \leq 4\}$.



Search state at iteration $n + 3$.

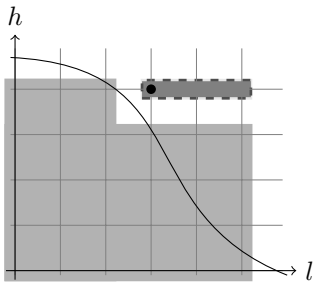
Since S contains a markable predicate for $U_{\mathcal{F}}$, the family $\mathcal{F}'$ is generated by adding S to $\mathcal{F}$. The predicate $\mu_S = (0 \leq l \leq 2)$ is selected as the marked predicate for that clause. It follows that

$$\mathcal{F}' = \left[\begin{array}{l} \{0 \leq l \leq 5, \underline{0 \leq h \leq 3}\} \\ \{\underline{0 \leq l \leq 2}, 0 \leq h \leq 4\} \end{array} \right]$$

and $U_{\mathcal{F}'} = \{0 \leq l \leq 5, 4 \leq h \leq 4, 3 \leq l \leq 5, 0 \leq h \leq 4\}$.

Iteration $n+4$.

To generate S, suppose the point $(3,4)$ is selected arbitrarily by `obstacle` in $X(U_{\mathcal{F}}) = \{(l,h) \in \mathbb{Z}^2 : 3 \leq l \leq 5, 4 \leq h \leq 4\}$. This point lies above Γ, and thus `obstacle` generates $S = \{3 \leq l \leq 5, 4 \leq h \leq 4\}$.



Search state at iteration $n+4$.

Since S contains no markable predicate for $U_{\mathcal{F}}$, we first generate R from the resolvents of S and the clauses in $\mathcal{F} = [C_1, C_2]$. Initiate the process with $R = S$ and $i = 2$.

$i = 2$**:** $\mu_2 = (0 \leq l \leq 2)$ implies that $\bar{\mu}_2 = (3 \leq l \leq 5)$, and this predicate is in R.
Replace R with $R \nabla C_2 = (R \setminus \{\bar{\mu}_2\}) \cup (C_2 \setminus \{\mu_2\}) = \{4 \leq h \leq 4, 0 \leq h \leq 4\}$.

$i = 1$**:** $\mu_1 = (0 \leq h \leq 3)$ implies that $\bar{\mu}_1 = (4 \leq h \leq 4)$, and this predicate is in R.
Replace R with $R \nabla C_1 = (R \setminus \{\bar{\mu}_1\}) \cup (C_1 \setminus \{\mu_1\}) = \{0 \leq h \leq 4, 0 \leq l \leq 5\}$.

The resulting nogood clause is $R = \{0 \leq h \leq 4, 0 \leq l \leq 5\}$.

Now determine the rank k such that R contains no markable predicate for $U_{\mathcal{F}_k}$. Initiate the process with $k = 0$ and $U_{\mathcal{F}_0} = \emptyset$.

$k = 0$**:** R contains no markable predicates for $U_{\mathcal{F}_0}$.

Thus $X(R) = \mathcal{X}$ and the search is completed. The optimum is $r^* = (3,3)$ with value $\bar{z} = -9$.

CHAPITRE 2

AN EXACT METHOD WITH VARIABLE FIXING FOR SOLVING THE GENERALIZED ASSIGNMENT PROBLEM

un article écrit par

Marius Posta[1,2], Jacques A. Ferland[1], Philippe Michelon[2]

et publié en ligne le 13 septembre 2011 dans

Computational Optimization and Applications.

Abstract

We propose a simple exact algorithm for solving the generalized assignment problem. Our contribution is twofold : we reformulate the optimization problem into a sequence of decision problems, and we apply variable-fixing rules to solve these effectively. The decision problems are solved by a simple depth-first lagrangian branch-and-bound method, improved by our variable-fixing rules to prune the search tree. These rules rely on lagrangian reduced costs which we compute using an existing but little-known dynamic programming algorithm.

1. Département d'Informatique et de Recherche Opérationnelle,
   Université de Montréal,
   C.P. 6128, succursale Centre-ville,
   Montréal, Québec, Canada, H3C 3J7.

2. Laboratoire d'Informatique d'Avignon,
   Université d'Avignon et des Pays du Vaucluse,
   84911 Avignon, Cedex 9, France.

2.1 Introduction

The Generalized Assignment Problem (GAP), originally specified by Ross and Soland [46], consists of assigning jobs to agents subject to resource constraints. Specifically, each job $j \in \{1, \ldots, n\}$ must be assigned to exactly one agent $i \in \{1, \ldots, m\}$. Let $\mathbf{c}_{ij}$ be the cost of assigning a job j to an agent i, let $\mathbf{a}_{ij}$ be the corresponding amount of resources consumed by that agent, and let $\mathbf{b}_i$ be that agent's resource capacity. These coefficients are all assumed to be integers. The mathematical model for (GAP) is as follows:

$$\begin{aligned}
\min \quad & \sum_{i=1}^{m}\sum_{j=1}^{n} \mathbf{c}_{ij}\mathbf{x}_{ij} & \\
\text{s.to} \quad & \sum_{j=1}^{n} \mathbf{a}_{ij}\mathbf{x}_{ij} \leq \mathbf{b}_i, & i = 1, \ldots, m; \ (1) \\
& \sum_{i=1}^{m} \mathbf{x}_{ij} = 1, & j = 1, \ldots, n; \ (2) \\
& \mathbf{x} \in \{0,1\}^{mn}. & (3)
\end{aligned}$$

Several exact search methods presented in the literature are branch-and-bound methods using the lagrangian relaxation obtained by dualizing the assignment constraints (2). If $\boldsymbol{\lambda}$ denotes the vector of multipliers, then this relaxation decomposes the problem into m 0-1 knapsack problems:

$$\begin{aligned}
z(\boldsymbol{\lambda}) = \sum_{j=1}^{n} \boldsymbol{\lambda}_j + \min \ & \sum_{i=1}^{m}\sum_{j=1}^{n} (\mathbf{c}_{ij} - \boldsymbol{\lambda}_j)\mathbf{x}_{ij} & \\
\text{s.to} \ & \sum_{j=1}^{n} \mathbf{a}_{ij}\mathbf{x}_{ij} \leq \mathbf{b}_i, & i = 1, \ldots, m; \ (1) \\
& \mathbf{x} \in \{0,1\}^{mn}. & (3)
\end{aligned}$$

The lagrangian dual bound $\max_{\boldsymbol{\lambda}} z(\boldsymbol{\lambda})$ thus obtained is at least as strong as the linear relaxation bound, which is equal to the lagrangian dual bounds obtained

by dualizing either the knapsack constraints, or both knapsack and assignment constraints, as shown in [49].

Both Haddadi and Ouzia [24] and Nauss [38] dualize the assignment constraints in their lagrangian branch-and-bound methods. The contribution of Haddadi and Ouzia [24] is a procedure for repairing the solutions of the lagrangian relaxation obtained during the subgradient search into primal-feasible solutions, thus maybe improving the upper bound. Nauss [38] presents a hybrid linear and lagrangian-relaxation-based method featuring knapsack cover cut generation. Savelsbergh [47] introduces a branch-and-price method, in which the columns generated correspond to feasible assignments for an agent. Pigatti et al. [41] accelerate the convergence of this method by proposing a scheme for stabilizing the multipliers during the column-generation phase. The method proposed by Avella et al. [2] is currently the best exact search method in the literature. It is a branch-and-cut method based on a knapsack cover cut generation scheme. Furthermore, several heuristic methods have been proposed: Yagiura et al. introduce a metaheuristic method based on an ejection chain neighborhood [50] [51] and Diaz and Fernandez [14] propose a simple and effective tabu search method.

In this paper we introduce a method in which the optimization problem is reduced to a sequence of decision problems. A valid lower bound is computed for (GAP) and then a search is performed for a feasible solution with the value of the lower bound. If none is found, the lower bound is incremented and the procedure is repeated. This continues until a feasible solution is found, which is then optimal. Naturally, our method can be efficient only if the gap between the initial lower bound and the optimal solution is small.

Section 2.2 includes an overview of our method. The variable-fixing procedure used by our algorithm is described in section 2.3. This procedure requires the reduced costs of the assignment variables. We compute these reduced costs using a dynamic programming approach based on results due to Karabakal et al. [31] summarized in section 2.4. Section 2.5 presents the numerical results obtained by running our implementation on the Beasley instance set [5]. The computation time

is often much less than that required by Avella et al. [2]. Furthermore, we provide previously unknown optimal values for three instances.

2.2 Solving a sequence of decision problems

Our approach can be outlined as follows:

- Denote by $\bar{z}$ a lower bound on the optimal value of the problem.
- Solve the following decision problem $GAP(\bar{z})$:
  - Does a feasible solution of cost $\bar{z}$ or less exist?
- If the answer is yes, then the solution is optimal, otherwise increment $\bar{z}$ by 1 and repeat.

This very crude algorithm relies on the fact that solution values are integer, and thus transforms the optimization problem into a sequence of decision problems. We use the lagrangian relaxation presented in the previous section to compute the initial lower bound, and also to compute the lower bounds for the nodes in the branch-and-bound method to solve the decision problems $GAP(\bar{z})$. Our branch-and-bound method to solve $GAP(\bar{z})$ can be outlined as follows:

1. Initialize the branch-and-bound active node queue with the root node.
2. If the active node queue is empty, then the answer to the decision problem is NO. Otherwise, select and remove a node from the queue.
3. Compute a lower bound for this node by solving the corresponding lagrangian dual. If, during the optimization of the dual, we obtain a relaxed solution which is primal-feasible and such that $z(\boldsymbol{\lambda}) \leq \bar{z}$, then the answer to the decision problem is YES, and the search is over.
4. If the lower bound exceeds the upper bound $\bar{z}$, go back to step 2.
5. Call the variable-fixing procedure.

6. Select the job j with the highest multiplier $\boldsymbol{\lambda}_j$ such that not all $\mathbf{x}_{ij}$ are fixed, and branch on each unfixed agent, thus creating up to m child nodes. Add the child nodes to the active node queue and go back to step 2.

Perhaps surprisingly, the numerical results in section 2.5 show that this approach works very well in practice. We can offer a few tentative explanations on why this is the case. To begin with, the lower bound provided by $\max_{\boldsymbol{\lambda}} z(\boldsymbol{\lambda})$ is usually good enough that the sequence of decision problems to be solved is very short.

Next, the variable-fixing procedure in step 5 is a key component of our algorithm since it significantly speeds up the resolution of the decision problems. Indeed it uses logical inference to fix variables to the value which is optimal in the current node for the current decision problem, given the global upper bound $\bar{z}$ and using information provided by the lagrangian relaxation. Obviously, a proper implementation of this procedure is critical to speed up the resolution of the successive decision problems. It is especially effective when the optimality gap is tight, and our decision-problem-sequence approach exploits this well. In the following sections, we present this procedure in detail.

Finally, current branch-and-bound methods and even the best metaheuristics often fail to identify good feasible solutions but we manage to bypass this problem entirely with our approach. Indeed, before developing the method presented in this paper, we tried to solve the optimization problem using a traditional branch-and-bound method (essentially that proposed by Haddadi and Ouzia [24]) enhanced with our variable-fixing rules. However, this was not improving the performance in all cases. Indeed, the variable-fixing rules were useless before reaching a good enough feasible solution, and for some instances finding a feasible solution is difficult. The method presented in this paper performs with a more consistent behavior. Additional details are included in our concluding analysis in subsection 2.5.4.

2.3 Variable fixing

As pointed out by Atamtürk and Savelsbergh in their survey [1], variable-fixing procedures are used in many linear-relaxation-based solvers. Our procedure exploits similar principles adapted to our lagrangian relaxation for (GAP). For any node of the branch-and-bound tree, denote by:

- $\boldsymbol{\lambda}^*$ the best multipliers found during the search for $\max_{\boldsymbol{\lambda}} z(\boldsymbol{\lambda})$[1],
- Δ the local optimality gap, i.e. $\Delta = \bar{z} - z(\boldsymbol{\lambda}^*)$,
- $\mathbf{x}(\boldsymbol{\lambda}^*) \in \{0,1\}^{mn}$ the optimal value of the lagrangian relaxation using the multipliers $\boldsymbol{\lambda}^*$.

Recall that the solution to the lagrangian relaxation is a binary vector. At any node, some components of this vector have been fixed to either 0 or 1, either through earlier branching or through earlier variable fixing. For any unfixed component $\mathbf{x}_{ij}$, if we fix $\mathbf{x}_{ij}$ to its value $\mathbf{x}(\boldsymbol{\lambda}^*)_{ij}$, then $z(\boldsymbol{\lambda}^*)$ does not change. However, if we fix $\mathbf{x}_{ij}$ to its complementary value $(1 - \mathbf{x}(\boldsymbol{\lambda}^*)_{ij})$, then $z(\boldsymbol{\lambda}^*)$ may increase. If it increases beyond $\bar{z}$ then we can conclude that no optimal solution can be found with $\mathbf{x}_{ij} = 1 - \mathbf{x}(\boldsymbol{\lambda}^*)_{ij}$, hence we may safely fix $\mathbf{x}_{ij}$ to the value $\mathbf{x}(\boldsymbol{\lambda}^*)_{ij}$.

Of course, computing $z(\boldsymbol{\lambda}^*)$ anew after fixing each unfixed component $\mathbf{x}_{ij}$ may be unreasonable from a computational point of view. Suppose however that we have $\mathbf{c}(\boldsymbol{\lambda}^*) \in \mathbb{R}^{mn}$, a vector of reduced costs associated to $\mathbf{x}(\boldsymbol{\lambda}^*)$. A reduced cost $\mathbf{c}(\boldsymbol{\lambda}^*)_{ij}$ associated to the variable $\mathbf{x}(\boldsymbol{\lambda}^*)_{ij}$ is a lower bound on the increase of the objective value of the relaxation when $\mathbf{x}_{ij}$ is forced to its complementary value $(1 - \mathbf{x}(\boldsymbol{\lambda}^*)_{ij})$. As a consequence, considering any unfixed variable $\mathbf{x}_{ij}$:

- if $\mathbf{x}(\boldsymbol{\lambda}^*)_{ij} = 0$ and $\mathbf{c}(\boldsymbol{\lambda}^*)_{ij} > \Delta$, then $\mathbf{x}_{ij}$ can be fixed to 0;
- if $\mathbf{x}(\boldsymbol{\lambda}^*)_{ij} = 1$ and $\mathbf{c}(\boldsymbol{\lambda}^*)_{ij} > \Delta$, then $\mathbf{x}_{ij}$ can be fixed to 1 and $\mathbf{x}_{kj}$ to 0 for all $k \neq i$.

[1] Thus the value $z(\boldsymbol{\lambda}^*)$ is a lower bound for this node

The preceding rules can be seen as 'simple' variable fixing rules. We may use more powerful rules by exploiting the fact that each job has to be assigned to one and only one agent, and that the lagrangian relaxation decomposes into m independent knapsack problems. Consider any unfixed variable $\mathbf{x}_{ij}$:

- Suppose that $\mathbf{x}(\boldsymbol{\lambda}^*)_{ij} = 0$ and $\sum_{k=1}^{m} \mathbf{x}(\boldsymbol{\lambda}^*)_{kj} \geq 1$:
  if $\mathbf{x}_{ij}$ were to take value 1 in a feasible solution, then all other $\mathbf{x}_{kj}$ would have to take value 0, therefore if

$$\mathbf{c}(\boldsymbol{\lambda}^*)_{ij} + \sum_{\substack{k=1 \\ \mathbf{x}(\boldsymbol{\lambda}^*)_{kj}=1}}^{m} \mathbf{c}(\boldsymbol{\lambda}^*)_{kj} > \Delta,$$

  then $\mathbf{x}_{ij}$ can be fixed to 0.

- Suppose that $\mathbf{x}(\boldsymbol{\lambda}^*)_{ij} = 1$ and $\mathbf{x}(\boldsymbol{\lambda}^*)_{kj} = 0$ for all $k \neq i$:
  if $\mathbf{x}_{ij}$ were to take value 0 in a feasible solution, then some $\mathbf{x}_{kj}$, $k \neq i$, not fixed to 0 would have to take value 1, therefore if

$$\mathbf{c}(\boldsymbol{\lambda}^*)_{ij} + \min\{\mathbf{c}(\boldsymbol{\lambda}^*)_{kj} \mid \mathbf{x}_{kj} \text{ unfixed}, k \in \{1, \ldots, i-1, i+1, \ldots, m\}\} > \Delta,$$

  then $\mathbf{x}_{ij}$ can be fixed to 1 and $\mathbf{x}_{kj}$ can be fixed to zero for all $k \neq i$.

- Suppose that $\mathbf{x}_{kj}$ is fixed to 0 for all $k \neq i$:
  obviously, $\mathbf{x}_{ij}$ can be fixed to 1. However if $\mathbf{x}(\boldsymbol{\lambda}^*)_{ij} = 0$ and $\mathbf{c}(\boldsymbol{\lambda}^*)_{ij} > \Delta$, then the current node can be fathomed.

Indeed we are sometimes able to strengthen the lower bound enough to fathom the current node. Considering now any job j:

- Suppose that $\sum_{i=1}^{m} \mathbf{x}(\boldsymbol{\lambda}^*)_{ij} = 0$:
  at least one variable $\mathbf{x}_{ij}$ for some $i \in \{1, \ldots, m\}$ must take value 1. Therefore if

$$\min\{\mathbf{c}(\boldsymbol{\lambda}^*)_{ij} \mid \mathbf{x}_{ij} \text{ unfixed}, i \in \{1, \ldots, m\}\} > \Delta,$$

  then the current node can be fathomed.

- Suppose that $\mathbf{x}_{ij}$ is fixed to 0 for all $i \in \{1, \ldots, m\}$:
  obviously, the current node can be fathomed.

Likely it is possible to derive yet more rules, however these work well enough: restricting ourselves to rules which fix variables to their current value in $\mathbf{x}(\boldsymbol{\lambda}^*)$ allows us to apply them successively to all unfixed variables without recomputing $\mathbf{c}(\boldsymbol{\lambda}^*)$ and $\mathbf{x}(\boldsymbol{\lambda}^*)$. In the following section, we describe how we compute the reduced costs $\mathbf{c}(\boldsymbol{\lambda}^*)$, upon which this variable fixing scheme depends.

2.4 Lagrangian reduced costs

The algorithm to compute the reduced costs is based on one introduced by Karabakal et al. in [31]. However, in [31] the output of this algorithm was used within a steepest-descent heuristic to solve the lagrangian dual $\max_{\boldsymbol{\lambda}} z(\boldsymbol{\lambda})$, and not to compute reduced costs to fix variables during the branch-and-bound search. Indeed, to our knowledge, we are the first to do so.

We now summarize how Karabakal et al. compute the reduced costs $\mathbf{c}(\boldsymbol{\lambda})$ given an arbitrary $\boldsymbol{\lambda}$. For the sake of simplicity and without loss of generality, we shall assume that we are at the root node, and that none of the variables $\mathbf{x}_{ij}$ have been fixed yet, however we apply the same procedure at every node of the branch-and-bound tree.

2.4.1 Decomposition of the lagrangian relaxation

Recall that when solving the lagrangian relaxation for a vector of multipliers $\boldsymbol{\lambda}$, we can decompose the problem into m independent knapsack subproblems. Denote by $\kappa^i(\boldsymbol{\lambda})$ the optimal value of the i-th knapsack subproblem associated with the multipliers $\boldsymbol{\lambda}$:

$$z(\boldsymbol{\lambda}) = \sum_{j=1}^{n} \boldsymbol{\lambda}_j + \sum_{i=1}^{m} \kappa^i(\boldsymbol{\lambda})$$

and

$$\begin{aligned} \kappa^i(\boldsymbol{\lambda}) = \min \ & \sum_{j=1}^{n} (\mathbf{c}_{ij} - \boldsymbol{\lambda}_j)\mathbf{x}_{ij} \\ \text{s.to} \ & \sum_{j=1}^{n} \mathbf{a}_{ij}\mathbf{x}_{ij} \leq \mathbf{b}_i, \\ & \mathbf{x}_{ij} \in \{0,1\}, \qquad \forall j \in \{1,\ldots,n\}. \end{aligned}$$

Denote by:

- $\mathbf{x}(\boldsymbol{\lambda})$ the optimal solution to the lagrangian relaxation with multipliers $\boldsymbol{\lambda}$,
- $\kappa^i(\boldsymbol{\lambda}, \mathbf{x}_{ij} = 0)$ the optimal value of the knapsack subproblem when a free variable $\mathbf{x}_{ij}$ is fixed to 0, and
- similarly $\kappa^i(\boldsymbol{\lambda}, \mathbf{x}_{ij} = 1)$ when $\mathbf{x}_{ij}$ is fixed to 1.

Fixing a free variable $\mathbf{x}_{ij}$ to its value in $\mathbf{x}(\boldsymbol{\lambda})$ will not change $z(\boldsymbol{\lambda})$, since by definition $\mathbf{x}(\boldsymbol{\lambda})$ is the optimal solution to the lagrangian relaxation given $\boldsymbol{\lambda}$. However fixing $\mathbf{x}_{ij}$ to its complement $(1-\mathbf{x}(\boldsymbol{\lambda})_{ij})$ may increase the objective value of the knapsack subproblem i from $\kappa^i(\boldsymbol{\lambda})$ to $\kappa^i(\boldsymbol{\lambda}, \mathbf{x}_{ij} = 1 - \mathbf{x}(\boldsymbol{\lambda})_{ij})$. We may therefore define the vector of lagrangian reduced costs $\mathbf{c}(\boldsymbol{\lambda})$ as the vector of these modifications:

$$\mathbf{c}(\boldsymbol{\lambda})_{ij} = \begin{cases} \kappa^i(\boldsymbol{\lambda}, \mathbf{x}_{ij} = 0) - \kappa^i(\boldsymbol{\lambda}) & \text{if } \mathbf{x}(\boldsymbol{\lambda})_{ij} = 1, \\ \kappa^i(\boldsymbol{\lambda}, \mathbf{x}_{ij} = 1) - \kappa^i(\boldsymbol{\lambda}) & \text{if } \mathbf{x}(\boldsymbol{\lambda})_{ij} = 0, \end{cases}$$

At this stage we already know $\kappa^i(\boldsymbol{\lambda})$ for all i, therefore computing $\mathbf{c}(\boldsymbol{\lambda})$ requires computing $\kappa^i(\boldsymbol{\lambda}, \mathbf{x}_{ij} = 1 - \mathbf{x}(\boldsymbol{\lambda})_{ij})$ for all (i,j). We now show how to do this efficiently using a dynamic programming approach.

2.4.2 Dynamic programming approach

Because the coefficients $\mathbf{a}_{ij}$ are all integer, we can solve the knapsack subproblem associated with each agent $i \in \{1,\ldots,m\}$ using a dynamic programming approach.

For each agent i, we define an initial state s^i, a final state t^i, and the states $v_\beta^{i,j}$ associated with each job $j \in \{1, \ldots, n\}$ and each possible resource usage $\beta \in \{0, \ldots, \mathbf{b}_i\}$.

We then define the following transition mechanism, given a vector of multipliers $\boldsymbol{\lambda}$. Consider any state $v_\beta^{i,j}$ with $1 \leq j < n$ and $0 \leq \beta \leq \mathbf{b}_i$:

- we allow transition to state $v_\beta^{i,j+1}$ at zero cost,
- we allow transition to state $v_{\beta+\mathbf{a}_{ij}}^{i,j+1}$ at cost $\mathbf{c}_{ij} - \boldsymbol{\lambda}_j$ if $\beta + \mathbf{a}_{ij} \leq \mathbf{b}_i$.

We also allow transition from s^i to $v_\beta^{i,1}$ as well as from $v_\beta^{i,n}$ to t^i for all $\beta \in \{0, \ldots, \mathbf{b}_i\}$ at zero cost. The state space for the knapsack problem for agent i parameterized by the multipliers $\boldsymbol{\lambda}$ can be represented as a weighted directed acyclic graph $G^i(\boldsymbol{\lambda})$ illustrated in figure 2.1.

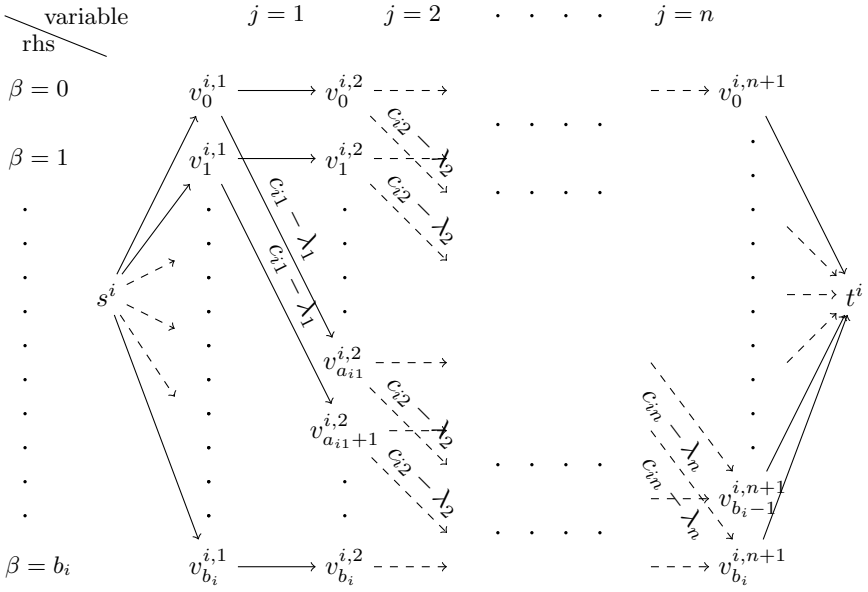


Figure 2.1: The graph $G^i(\boldsymbol{\lambda})$.

Representing the knapsack subproblem as a shortest-path problem lets us see that the optimal value $\kappa^i(\boldsymbol{\lambda})$ is the length of the shortest path from s^i to t^i in $G^i(\boldsymbol{\lambda})$. Denote by $f^i(\boldsymbol{\lambda}, j, \beta)$ the length of the shortest path from s^i to $v_\beta^{i,j}$, and

by $g^i(\boldsymbol{\lambda}, j, \beta)$ the length of the shortest path from $v_\beta^{i,j}$ to t^i. Following the Bellman optimality principle, these values can be expressed recursively:

$$f^i(\boldsymbol{\lambda}, j, \beta) = \begin{cases} 0 & \text{if } j = 1, \\ f^i(\boldsymbol{\lambda}, j-1, \beta) & \text{if } \beta < \mathbf{a}_{ij}, \\ \min \left\{ \begin{array}{c} f^i(\boldsymbol{\lambda}, j-1, \beta), \\ f^i(\boldsymbol{\lambda}, j-1, \beta - \mathbf{a}_{ij}) + \mathbf{c}_{ij} - \boldsymbol{\lambda}_j \end{array} \right\} & \text{otherwise,} \end{cases}$$

$$g^i(\boldsymbol{\lambda}, j, \beta) = \begin{cases} 0 & \text{if } j = n+1, \\ g^i(\boldsymbol{\lambda}, j+1, \beta) & \text{if } \beta > \mathbf{b}_i - \mathbf{a}_{ij}, \\ \min \left\{ \begin{array}{c} g^i(\boldsymbol{\lambda}, j+1, \beta), \\ g^i(\boldsymbol{\lambda}, j+1, \beta + \mathbf{a}_{ij}) + \mathbf{c}_{ij} - \boldsymbol{\lambda}_j \end{array} \right\} & \text{otherwise.} \end{cases}$$

Notice that we may obtain $\kappa^i(\boldsymbol{\lambda})$ by computing either $f^i(\boldsymbol{\lambda}, n, \mathbf{b}_i)$ or $g^i(\boldsymbol{\lambda}, 0, 0)$ using a dynamic programming approach.

We now show how this approach also leads itself to solving the knapsack problem for agent i given the multipliers $\boldsymbol{\lambda}$ in which a variable $\mathbf{x}_{ij}$ has been fixed to a specific value, i.e. $\kappa^i(\boldsymbol{\lambda}, \mathbf{x}_{ij} = 0)$ or $\kappa^i(\boldsymbol{\lambda}, \mathbf{x}_{ij} = 1)$. Fixing a variable $\mathbf{x}_{ij}$ to 1 consists in removing a subset of arcs from the state space $G^i(\boldsymbol{\lambda})$, namely the arcs $(v_\beta^{i,j}, v_\beta^{i,j+1})$ for all $\beta \in \{0, \ldots, \mathbf{b}_i\}$. Similarly, fixing a variable $\mathbf{x}_{ij}$ to 0 consists in removing from $G^i(\boldsymbol{\lambda})$ all arcs $(v_\beta^{i,j}, v_{\beta+\mathbf{a}_{ij}}^{i,j+1})$ with $\beta \in \{0, \ldots, \mathbf{b}_i\}$. Fortunately, the graph $G^i(\boldsymbol{\lambda})$ is topologically ordered, and it is relatively easy to compute the length of the new shortest path. Indeed, according to the Bellman optimality principle, we have

$$\kappa^i(\boldsymbol{\lambda}, \mathbf{x}_{ij} = 0) = \min_{0 \leq \beta \leq \mathbf{b}_i} \left(f^i(\boldsymbol{\lambda}, j, \beta) + g^i(\boldsymbol{\lambda}, j+1, \beta) \right).$$

and similarly,

$$\kappa^i(\boldsymbol{\lambda}, \mathbf{x}_{ij} = 1) = \mathbf{c}_{ij} - \boldsymbol{\lambda}_j + \min_{0 \leq \beta \leq \mathbf{b}_i - \mathbf{a}_{ij}} \left(f^i(\boldsymbol{\lambda}, j, \beta) + g^i(\boldsymbol{\lambda}, j+1, \beta + \mathbf{a}_{ij}) \right).$$

As a consequence the reduced costs $\mathbf{c}(\boldsymbol{\lambda})$ can be found in $O(mn\mathbf{b}_i)$.

The case where a variable $\mathbf{x}_{ij'}$ is fixed to 1 is illustrated in figure 2.2. The arcs $(v_\beta^{i,j'}, v_\beta^{i,j'+1})$ are removed from $G^i(\boldsymbol{\lambda})$ for all $\beta \in \{0, \ldots, \mathbf{b}_i\}$. If we assume that the shortest path marked in full thick lines corresponds to the value of $\kappa^i(\boldsymbol{\lambda})$, then to obtain $\kappa^i(\boldsymbol{\lambda}, \mathbf{x}_{ij'} = 1)$ we need to examine each arc $(v_\beta^{i,j'}, v_{\beta+\mathbf{a}_{ij'}}^{i,j'+1})$ for all $\beta \in \{0, \ldots, \mathbf{b}_i - \mathbf{a}_{ij'}\}$ to compute the new shortest path (marked in dashed thick lines).

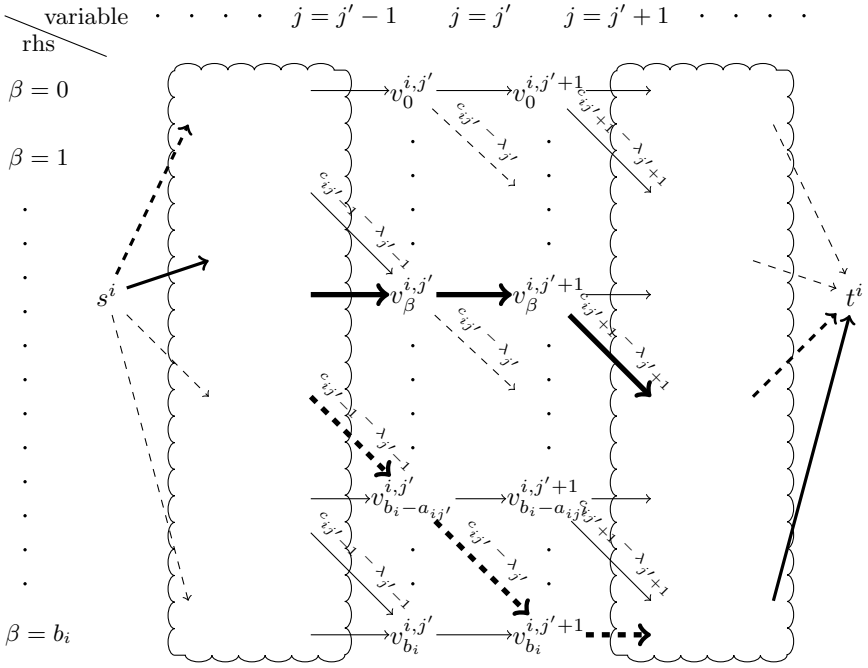


Figure 2.2: The new shortest path after fixing $\mathbf{x}_{ij'}$ to 1.

2.5 Computational experience

First we introduce a few details pertaining to the implementation of our method which warrant further discussion. We then specify the experimental protocol applied for obtaining our numerical results. These are then presented and compared with the best known results. Finally we conclude by analyzing the efficiency of our variable fixing procedure.

2.5.1 Implementation details

The efficient resolution of knapsack problems in the implementation of our method is a matter of great importance. These knapsack problems result from the decomposition of the lagrangian relaxation of (GAP), and in our method this relaxation is used in two places: in the resolution of the lagrangian dual, and in the computation of lagrangian reduced costs. In the latter case we use dynamic programming as explained previously in subsection 2.4.2, however in the former we use a more sophisticated algorithm, MINKNAP, developed by David Pisinger [42]. This choice proved essential to solving the difficult problem instances in which the coefficients are strongly correlated. Interestingly, Avella et al. [2] who obtained the best known results also use MINKNAP for their knapsack cover cut generation. Until now in the generalized assignment problem literature, such knapsack problems were solved either using a simple branch-and-bound or even using dynamic programming.

Also until now, it seems that previous authors using lagrangian relaxations solved the lagrangian dual with a subgradient method. We choose instead to solve it with a bundle method, and more specifically using Antonio Frangioni's [B]TT/OBP solver [17]. The bundle method is robust, requiring fewer parameter adjustments than the subgradient method to perform acceptably. In fact we found that allowing up to around a hundred bundle iterations worked well enough for all problems in our instance set, and did not tweak the method further. On the contrary, despite all our efforts in tweaking the magic numbers for the subgradient method, we never came close to performing as well as the bundle method. Another motivation for preferring the bundle method is that in solving $\max_{\boldsymbol{\lambda}} z(\boldsymbol{\lambda})$, the computational bottleneck is clearly in solving $z(\boldsymbol{\lambda})$ for a given $\boldsymbol{\lambda}$, rather than in updating and optimizing the bundle model of $\{z(\boldsymbol{\lambda}) \mid \boldsymbol{\lambda} \in \mathbb{R}^n\}$.

When solving a decision problem $GAP(\bar{z})$, we naturally choose to use a depth-first node selection strategy for the branch-and-bound search, since we wish to find a feasible solution as soon as possible. If there are none, the node selection strategy

has no impact other than on memory use, which fortunately is minimized when going depth-first. Also, when evaluating the lower bound in all nodes (except at the root node), the bundle method is initialized with the best multipliers found for the parent node, and is run until a fixed iteration limit is reached (around 100 works well), or until the bundle method performs a minor step of size below a certain threshold (10^{-5}).

For the root node, we re-use the multipliers obtained during the computation of the initial lower bound (which also yields the initial $\bar{z}$), thus avoiding some replication of effort. Note that it is possible to extend this approach to other nodes of the search tree where no variables have been fixed other than by branching, since we find these nodes in the search tree for the next decision problem $GAP(\bar{z}+1)$. We have not done this, because we noticed during our tests that the variable-fixing procedure is effective very early on in the search, in most cases fixing variables in all nodes of a search tree.

We compute the initial lower bound as follows. First we solve the linear relaxation of the MIP model of the instance with CPLEX. We then initialize the bundle method with the optimal knapsack constraint multipliers of the linear relaxation, and we let it run with a much higher iteration limit (around 100,000).

2.5.2 Experimental protocol

The best known results for these problems are provided by Avella et al. [2]. They compare thoroughly and favorably their results with the previously best known results in literature. As most works in the literature, we solve the instances of the Beasley set [5], which are divided into 5 categories, A B C D and E. We ignore the instances in categories A and B as Avella et al. have done, because they are too easy. In our tables the instances are named ymn, where y designates the category, m the number of agents and n the number of jobs.

The data which interests us is the execution duration of our program, for the full search as well as for the optimality proof. A full search corresponds to the execution of our program without prior knowledge of any feasible solution value.

An optimality proof corresponds to showing there are no solutions with a lower value than the best known. In other words, given a best known solution value z^* (as reported in the literature), to prove optimality our program needs to solve the decision problem $GAP(z^* - 1)$, and Avella et al. need to explore their branch-and-cut search tree using $z^* - 1$ as a cutoff value.

The computer used by Avella et al. is a Pentium IV CPU clocked at 3.2 GHz. For our experiments, we had at our disposal a group of cluster nodes all equipped with identical dual AMD Opteron 246 processors. Since Avella et al. also provide the computation time to solve the problems with the CPLEX solver (version 10.1), we did the same to conservatively estimate our machines to be twice as fast as theirs, which concurs with number crunching benchmarks published on a serious computer hardware website (specifically: tomshardware.com). Our algorithm is coded in the C language, but since we use C++ code from [B]TT, we compiled it all with `g++` version 4.1 with all optimizations enabled and targeting a x86-64 architecture. Our program executions were all limited to 24 hours user time.

2.5.3 Results

Table 2.I presents the best lower bounds computed by our program for each instance. The initial lower bound is obtained when solving the lagrangian dual in order to determine the initial value of $\bar{z}$. The two next columns list the global lower bounds on the optimal value obtained after 30 minutes and 24 hours of execution time of our method. If no value is indicated in these columns, then the program found an optimal solution before the corresponding time limit. These optimal values are listed in 'Optimal', and are highlighted in bold if they were previously unknown. Otherwise, for the instances which were not solved to optimality in 24 hours, we provide the best known feasible solution values found in the literature in the next column 'Best known'.

Table 2.II presents the execution times in seconds for our program and Avella et al. [2]. The total number of nodes evaluated and the total number of lagrangian relaxations solved until finding an optimal solution are lised in the columns 'Nodes'

and 'Relaxations', respectively, for the instances for which this happened within 24 hours. The results of Avella et al. are indicated as in [2], i.e. without accounting for the difference in computing power. We removed from the table the 10 instances for which neither our program or theirs could even perform an optimality proof. Nine out of these ten are instances of the D category, characterized by having tight knapsack constraints in which the weights are strongly correlated to the prices. The improvement factor is the ratio of their time over our time, divided by 2 to conservatively allow for the difference in hardware, for both full searches and optimality proofs.

As can be seen, in almost all cases our results are much better than theirs, which were state-of-the-art. We find three previously unknown optimal solutions, and are able to find the optimal solution of an instance in all cases they are able to, save one. Our full searches all perform faster and are better than theirs. There are only five instances for which Avella et al. obtain optimality proofs faster than we do, and there is only one instance for which we fail to find an optimal solution while they succeed.

2.5.4 Concluding analysis

Interestingly, three of the five instances for which Avella et al. perform better to prove optimality are the largest instances of the C category: c40400, c60900 and c401600. The other instances are e201600, a large instance, and c10200, a small but easy instance. The instances in categories C and E are also less constrained than those in category D having tight knapsack constraints whose coefficients are highly correlated to the assignment prices. As a consequence even finding good feasible solutions for instances in category D can be a challenge, notice however that our method performs relatively well in D.

The curves in figure 2.3 illustrate the benefit of using our variable-fixing scheme. The points on FULL indicate the cumulative time spent for solving the decision problems up to the current value for $\bar{z}$, when using all variable fixing rules presented in section 2.3. The points on SIMPLE correspond to applying only the so-called

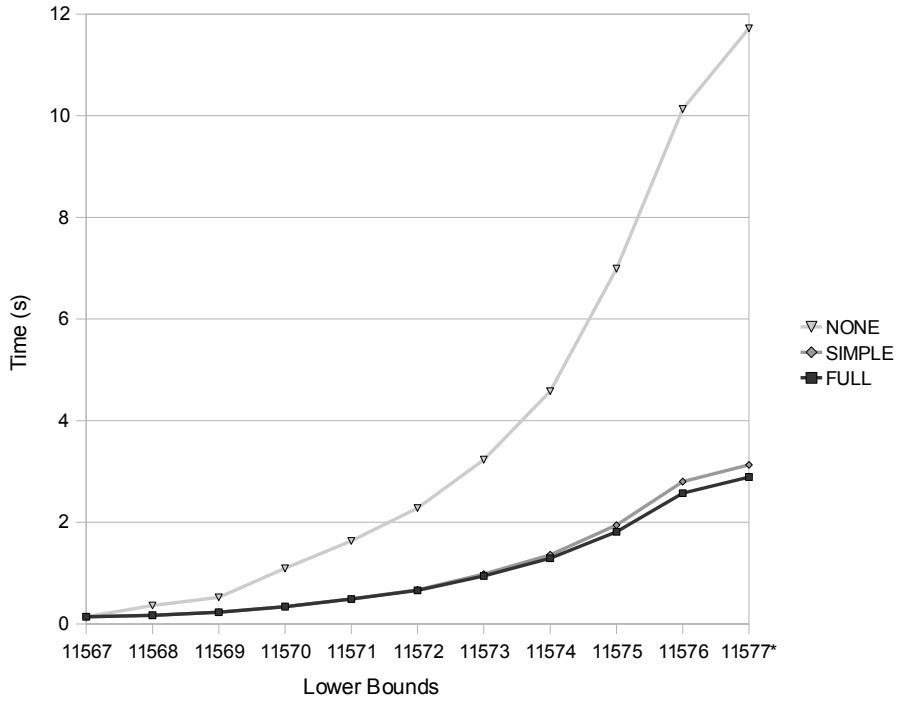


Figure 2.3: Impact of variable fixing on the time required for instance e10100.

simple variable fixing rules, and NONE to applying no variable fixing rules at all.

This illustrates that computing the lagrangian reduced costs and applying at least the simple rules is strongly beneficial to the overall performance of the search. Applying the full set of rules incurs little overhead once the reduced costs have been computed, and while the improvement over the simple rules is not in the same order of magnitude, it still appears to be worthwile.

The results for this 10-agent 100-job instance are representative of the other instances, for which a similar pattern emerges. The cumulative time also strongly correlates with the cumulative branch-and-bound tree node count as well as the cumulative number of lagrangian relaxations evaluated. This is the case for this instance as well as for the others.

Now let us come back to our claim made at the end of section 2.2 that our method performs more consistently than more straightforward branch-and-bound methods. We initially implemented a method very similar to that proposed by Had-

dadi and Ouzia [24], namely a lagrangian branch-and-bound where an additional repair process is applied at each node to reach feasible solutions and improve the upper bound, to which we added our variable-fixing rules. However, we observed that the performance of such a method is very sensitive to how soon a good feasible solution is found. In particular, we obtained very uneven results for the instances in the D category, where the tight knapsack constraints make it difficult to find feasible solutions.

Figure 2.4 illustrates this for instance d05100 (5 agents, 100 jobs), the optimum being 6353. We plot the evolution of the upper and lower bounds of three methods: our implementation of Haddadi and Ouzia [24] in light gray, this method with the addition of our variable-fixing rules in dark gray, and the method presented in this paper in black. Notice that the use of variable-fixing rules has hardly any impact on the lower bound in the straighforward branch-and-bound implementation, in contrast to our method presented in this paper (also illustrated previously in figure 2.3).

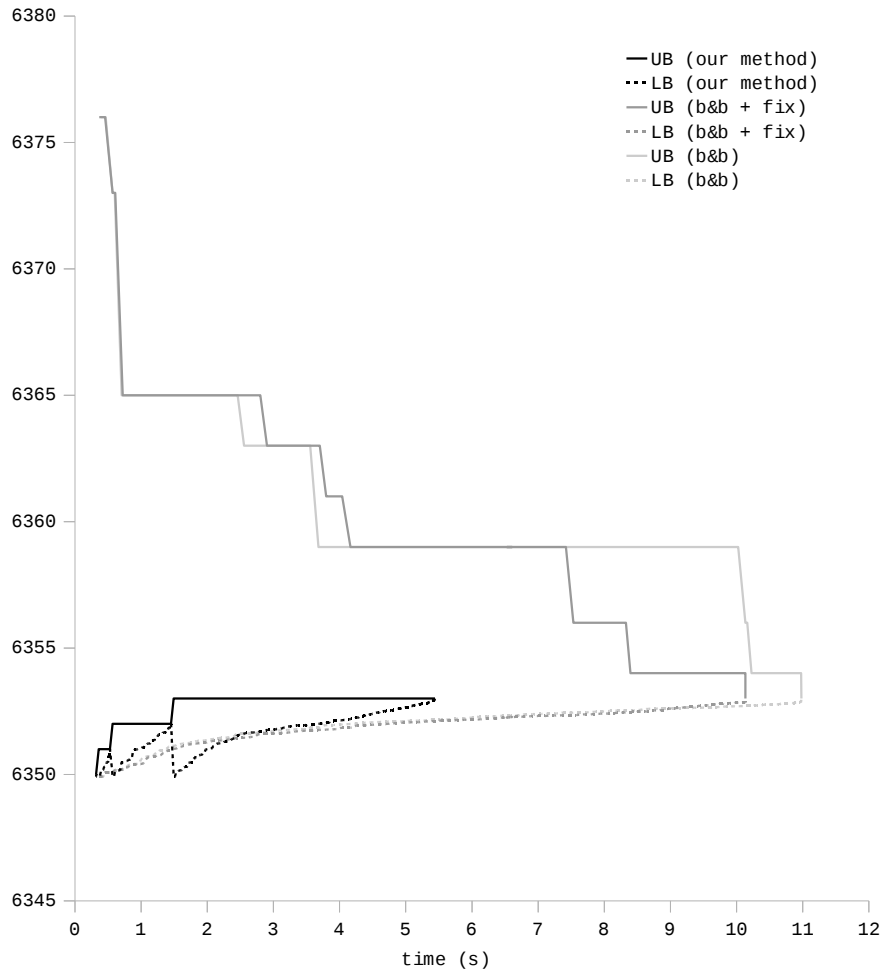


Figure 2.4: Evolution of bound values for instance d05100.

Let us conclude with a few additional comments. Solving the instance d05100 with our method without using any variable-fixing rules requires about 18 seconds, and this is consistent with the behavior illustrated in figure 2.3. Note also that the time reported in table 2.II for solving this instance is 5.07 seconds, which is slightly less than depicted in figure 2.4. The reason for this apparent discrepancy is that we used best-first node selection in figure 2.4 instead of depth-first.

Finally, notice that if we increment $\bar{z}$ by a value ϵ instead of 1, then without any further modifications our method produces an ϵ-optimal solution, which may be of

interest in the case where the coefficients $\mathbf{c}_{ij}$ are fractionary. It may be interesting to find other 0-1 problems for which a scheme similar to ours could be succesfully applied.

Acknowledgements

We wish to thank the associate editor and the anonymous referees for their comments and suggestions which helped improve the clarity of this paper.

Table 2.I: Solution values found by our method.

Instance	Initial LB	LB (1/2h+)	LB (24h+)	Optimal	Best known
c05100	1930			1931	
c05200	3455			3456	
c10100	1400			1402	
c10200	2804			2806	
c10400	5596			5597	
c15900	11339			11340	
c20100	1242			1243	
c20200	2391			2391	
c20400	4781			4782	
c201600	18802	18802		**18802**	
c30900	9982			9982	
c40400	4244			4244	
c401600	17144	17144	17145		17145
c60900	9325	9326	9326		9326
c801600	16284	16284	16284		16289
d05100	6350			6353	
d05200	12741			12742	
d10100	6342			6347	
d10200	12426			12430	
d10400	24959			**24961**	
d15900	55403	55403	55404		55414
d20100	6177	6184		6185	
d20200	12230	12234	12235		12244
d20400	24561	24562	24563		24585
d201600	97823	97824	97824		97837
d30900	54833	54833	54834		54868
d40400	24350	24350	24350		24417
d401600	97106	97106	97106		97113
d60900	54551	54551	54551		54606
d801600	97034	97034	97034		97052
e05100	12673			12681	
e05200	24927			24930	
e10100	11568			11577	
e10200	23302			23307	
e10400	45745			45746	
e15900	102420			102421	
e20100	8432			8436	
e20200	22377			22379	
e20400	44876			44877	
e201600	180644			180645	
e30900	100427			100427	
e40400	44557			44561	
e401600	178292			178293	
e60900	100147	100148		100149	
e801600	176819			**176820**	

Table 2.II: Performance.

	Our program				Avella et al.		Improvement factor	
Instance	Nodes	Relaxations	Search	Proof	Search	Proof	Search	Proof
c05100	6	397	.05	.04	2.39	.60	23.9	7.5
c05200	11	2,112	.42	.36	9.03	1.50	10.8	2.1
c10100	31	1,543	.16	.13	2.67	.80	8.3	3.1
c10200	248	17,970	3.12	2.28	17.13	2.90	2.7	.6
c10400	157	13,898	4.54	2.65	38.61	5.70	4.3	1.1
c15900	406	26,894	17.27	2.70		2,257.37		418.0
c20100	50	2,226	.29	.17	1.83	.80	3.2	2.4
c20200	52	3,689	.81	.17	13.98	1.80	8.6	5.3
c20400	1,034	63,210	25.32	9.79	91.52	21.90	1.8	1.1
c201600	62,630	3,759,201	5,804.55	3.52				
c30900	6,826	369,235	373.27	3.35	3,997.56	12.30	5.4	1.8
c40400	253	16,626	9.94	2.87	52.38	3.30	2.6	.6
c401600				11,831.26		3,231.14		.1
c60900				203.99	8,369.69	121.40		.3
d05100	617	36,848	5.07	1.28	17.08	13.30	1.7	5.2
d05200	159	10,535	2.33	1.24	17.83	13.80	3.8	5.6
d10100	3,824	191,735	41.33	13.37		385.29		14.4
d10200	58,840	3,346,652	1,144.67	417.02		8,921.75		10.7
d10400	19,577	1,157,941	513.14	174.43				
d20100	379,632	13,857,852	4,010.77	2,425.81		27,783.04		5.7
e05100	231	14,013	1.05	.28	5.33	2.00	2.5	3.6
e05200	82	5,389	.59	.23	5.69	1.50	4.8	3.3
e10100	425	23,040	2.92	.90	8.94	4.70	1.5	2.6
e10200	441	27,903	4.91	2.92	39.41	7.40	4.0	1.3
e10400	27	2,385	.86	.56	98.91	3.50	57.5	3.1
e15900	67	4,573	3.58	.95	203.45	6.60	28.4	3.5
e20100	207	10,788	2.53	1.21	51.14	8.30	10.1	3.4
e20200	34	2,662	.98	.65	42.89	5.80	21.9	4.5
e20400	79	4,071	1.97	.94	79.89	8.10	20.3	4.3
e201600	386	24,803	40.02	35.80	1,003.47	32.70	12.5	.5
e30900	71	3,743	4.56	.77	671.51	13.20	73.6	8.6
e40400	3,223	143,889	104.59	56.58	2,875.01	636.60	13.7	5.6
e401600	1,631	84,549	243.02	3.69	4,123.61	54.60	8.5	7.4
e60900	210,933	9,519,630	23,181.30	1,360.24		6,341.15		2.3
e801600	285	14,230	75.51	12.05				

CHAPITRE 3

AN EXACT COOPERATIVE METHOD FOR THE UNCAPACITATED FACILITY LOCATION PROBLEM

un article écrit par

Marius Posta[1,2], Jacques A. Ferland[1], Philippe Michelon[2]

et bientôt soumis à une revue scientifique avec comité de lecture.

Abstract

In this paper, we present a cooperative primal-dual method to solve the uncapacitated facility location problem exactly. It consists of a primal process, which performs a variation of a known and effective tabu search, and a dual process, which performs a lagrangian branch-and-bound search. Both processes cooperate by exchanging information which helps them find the optimal solution. Further contributions include new techniques for improving the evaluation of the branch-and-bound nodes : decision-variable bound tightening rules applied at each node, and a subgradient caching strategy to improve the bundle method applied at each node.

1. Département d'Informatique et de Recherche Opérationnelle,
   Université de Montréal,
   C.P. 6128, succursale Centre-ville,
   Montréal, Québec, Canada, H3C 3J7.

2. Laboratoire d'Informatique d'Avignon,
   Université d'Avignon et des Pays du Vaucluse,
   84911 Avignon, Cedex 9, France.

3.1 Introduction

The Uncapacitated Facility Location Problem is a well-known combinatorial optimization problem, also known as the Warehouse Location Problem and as the Simple Plant Location Problem. The problem consists in choosing among n locations where plants can be built to service m customers. Building a plant at a location $i \in \{1, \ldots, n\}$ incurs a fixed opening cost $\mathbf{c}_i$, and servicing a customer $j \in \{1, \ldots, m\}$ from this location incurs a service cost $\mathbf{s}_{ij}$. The objective is to minimize the total cost:

$$\min_{\mathbf{x} \in \{0,1\}^n} f(\mathbf{x}), \quad \text{with } f(\mathbf{x}) = \sum_{i=1}^{n} \mathbf{c}_i \mathbf{x}_i + \sum_{j=1}^{m} \min \left\{ \mathbf{s}_{ij} : \mathbf{x}_i = 1, i \in \{1, \ldots, n\} \right\}.$$

Note that all solutions in $\{0,1\}^n$ are feasible except for $\mathbf{0}$. The problem can be formulated as a 0-1 mixed integer programming problem. This requires the introduction of a vector of auxiliary variables $\mathbf{y} \in [0,1]^{nm}$ in which each component $\mathbf{y}_{ij}$ represents the servicing of customer j by location i. The following model is the so-called 'strong' formulation, denoted (P):

$$\begin{aligned}
\min \quad & \sum_{i=1}^{n} \mathbf{c}_i \mathbf{x}_i + \sum_{j=1}^{m} \sum_{i=1}^{n} \mathbf{s}_{ij} \mathbf{y}_{ij} && && (P) \\
\text{s.to} \quad & \sum_{i=1}^{n} \mathbf{y}_{ij} = 1, && 1 \leq j \leq m, && (1) \\
& 0 \leq \mathbf{y}_{ij} \leq \mathbf{x}_i, && 1 \leq i \leq n, 1 \leq j \leq m, && (2) \\
& \mathbf{x} \in \{0,1\}^n. && && (3)
\end{aligned}$$

Note that this model has at least one optimal solution in which $\mathbf{y}$ is integer.

Much work on the UFLP has been published during the last decade, mainly various heuristic methods [12, 16, 19, 23, 26, 29, 30, 33, 35, 45, 48]. In contrast, most publications on the exact resolution of this problem are centered on the dual ascent methods proposed by Erlenkotter [15] and by Bilde & Krarup [8]. The `DualLoc` method of Erlenkotter, subsequently improved by Körkel [32] and recently

by Letchford & Miller [34], is a branch-and-bound method in which a lower bound at each node is evaluated by a dual ascent heuristic (see also [36] for a study of its optimality gap).

A different method which was proposed recently is the semi-lagrangian approach of Beltran-Royo et al. [7], originally developed for solving the k-median problem [6]. Other methods include the primal-dual method proposed by Galvão & Raggi [18], which combines a primal procedure to find a good solution and a dual procedure to close the optimality gap. The heuristic proposed by Hansen et al. [26] can be adapted to solve the UFLP exactly. On a related note, Goldengorin et al. [21, 22] propose some preprocessing rules to improve branch-and-bound search methods applied to the UFLP. These rules analyze the relationships between the opening costs $\mathbf{c}$ and the service costs $\mathbf{s}$ to try to determine the optimal state of each location. However, many of the more difficult instances in the `UflLib` collection are purposely designed to have a contrived cost structure on which few deductions can be made. General purpose MIP solvers such as CPLEX struggle to solve these instances, in many cases spending a very long time just to evaluate the root node. Indeed, the constraints (2) make the linear relaxation of (P) difficult to solve using simplex algorithms because of degeneracy issues. There are also mn of them, and for many instances in `UflLib` we have $n = m = 1000$. It is possible to aggregate these constraints into $\sum_{j=1}^{m} \mathbf{y}_{ij} \leq m\mathbf{x}_i$ for all $i \in \{1, \ldots, n\}$, but the bound induced by the corresponding linear relaxation is too weak to be of any practical use, even when strengthened with cuts.

We propose a new method for solving (P) exactly, in which a metaheuristic and a branch-and-bound cooperate to reduce the optimality gap. In this respect, this is somewhat similar to the work of Galvão & Raggi [18], however our method is *co-operative*: the metaheuristic and the branch-and-bound exchange information such that they perform better than they would in isolation. Information is exchanged by message-passing between two processes. Each message has a sender, a receiver, a send date, a type and maybe some additional data. Each process sends and receives messages asynchronously at certain specific moments during their execution, and

if at any of these moments there are several incoming messages of the same type, then all but the most recent are discarded. Similar approaches have been proposed before, a recent example is the work of Muter et al. [37] on solving the set-covering problem, but to our knowledge this paper introduces the first cooperative approach specifically for solving the UFLP exactly.

Section 3.2 describes the *primal process*, which improves an upper bound by searching for solutions through a metaheuristic. Section 3.3 describes the *dual process*, which improves a lower bound by enumerating a branch-and-bound search tree, using lower bounds derived from the lagrangian relaxation obtained by dualizing constraints (1). Contrary to most work in the literature which use dual-ascent heuristics (e.g., `DualLoc`), we use a bundle method to optimize the corresponding lagrangian dual. Additional contributions of this paper include new strategies for partitioning (P), as well as a subgradient caching technique to improve the performance of bundle method in the context of our branch-and-bound.

Section 3.4 concludes this paper by presenting our computational results. Our method is very effective on almost all UFLP instances in `UflLib`, and solved many to optimality for the first time. These new optimal values can be found in the appendix, along with several implementation details which do not enrich this paper.

3.2 Primal process

The purpose of the primal process is to find good solutions for (P). Specifically, we use a variant of the tabu search proposed by Michel & Van Hentenryck [35] which we shall now briefly describe. This local search method explores the 1OPT neighborhood of a given solution $\check{\mathbf{x}} \in \{0,1\}^n$. This neighborhood is defined as the set of solutions obtained by flipping one component of $\check{\mathbf{x}}$: either opening a closed location, or closing an open location.

Denote $\check{\mathbf{x}}^i$ the neighbor of $\check{\mathbf{x}}$ obtained by flipping its i-th component. For all components $i \in \{1,\ldots,n\}$, denote $\boldsymbol{\delta}_i = f(\check{\mathbf{x}}^i) - f(\check{\mathbf{x}})$ the transition cost associated with moving from $\check{\mathbf{x}}$ to $\check{\mathbf{x}}^i$. Michel & Van Hentenryck [35] show how to update

the vector of transition costs $\boldsymbol{\delta}$ in $O(m \log n)$ time after each move in the 1OPT neighborhood (see also Sun [48]). They argue that this efficient evaluation of the neighborhood is the main reason why their local search scheme performs as well as it does. The basic idea is to maintain the open locations in a heap for each customer $j \in \{1, \ldots, m\}$, ordered in increasing supply costs $\mathbf{s}_{ij}$: when opening a closed location add this location inside each heap, and when closing an open location remove this location from each heap.

Our tabu search is a variant of that presented in [35]. Our main modification is that it allows some solution components to be fixed to their current values by having a tabu status with infinite tenure. Tabu status is stored in an array $\boldsymbol{\tau}$ of dimension n, where each component can take any value in $\mathbb{N} \cup \{\infty\}$ specifying the iteration at which the tabu status expires.

At each iteration, the search performs a 1OPT-move on a location $w \in \{1, \ldots, n\}$ selected as follows. The location w is the one decreasing the objective function the most. Furthermore the move should not be tabu or it should satisfy the aspiration criterion. If no such move exists, then we perform a diversification by selecting w at random among all non-tabu locations. If none of these exist, then we select w at random among all non-fixed locations. If none of these exist either, then there is nothing left to do and we end the search.

We modify the tabu status $\boldsymbol{\tau}_w$ according to the current tenure variable `tenure`, which indicates the number of subsequent iterations during which $\check{\mathbf{x}}_w$ cannot change. The value of `tenure`, initially set to 10, remains within the interval $[2, 10]$. During the search it is decreased by 1 following each improving move (if `tenure` > 2) and it is increased by 1 following each non-improving move (if `tenure` < 10).

Before performing the next iteration, the primal process performs its communication duties, dealing with incoming messages and sending outgoing messages to the dual process. Although this tabu search method works well enough for finding good solutions [35] on its own, communication with the dual process nonetheless allows us (among other things) to fix some solution components to their optimal values, thereby narrowing the search space. The following list specifies the kinds

of messages our primal process sends and receives:

- Outgoing:
  - send the incumbent solution $\bar{\mathbf{x}}^{\texttt{primal}}$,
  - send a request for a *guiding solution*.
- Incoming:
  - receive a solution $\bar{\mathbf{x}}^{\texttt{dual}}$,
  - receive an *improving partial solution* $\bar{\mathbf{p}}$,
  - receive a guiding solution $\underline{\mathbf{x}}$.

Let $\bar{\mathbf{x}}^{\texttt{primal}}$ be the best solution known by the primal process. Whenever $\check{\mathbf{x}}$ is such that $f(\check{\mathbf{x}}) < f(\bar{\mathbf{x}}^{\texttt{primal}})$, the primal process sets $\bar{\mathbf{x}}^{\texttt{primal}}$ to $\check{\mathbf{x}}$ and sends it to the dual process as soon as possible. The dual process likewise maintains its own incumbent solution, denoted $\bar{\mathbf{x}}^{\texttt{dual}}$, and sends it to the primal process as soon as it is improved. Whenever the primal process receives $\bar{\mathbf{x}}^{\texttt{dual}}$ such that $f(\bar{\mathbf{x}}^{\texttt{dual}}) < f(\bar{\mathbf{x}}^{\texttt{primal}})$, the primal process sets $\bar{\mathbf{x}}^{\texttt{primal}}$ to $\bar{\mathbf{x}}^{\texttt{dual}}$.

A guiding solution $\underline{\mathbf{x}} \in \{0,1\}^n$ is used to perturb $\boldsymbol{\tau}$ such that a move involving any location $i \in \{1,\ldots,n\}$ such that $\check{\mathbf{x}}_i = \underline{\mathbf{x}}_i$ becomes tabu. As a consequence, $\check{\mathbf{x}}$ moves closer to $\underline{\mathbf{x}}$ during the next iterations. This is similar in spirit to path-relinking [20].

An improving partial solution $\bar{\mathbf{p}} \in \{0,1,*\}^n$ defines a solution subspace which is known to contain all solutions cheaper than $\bar{\mathbf{x}}^{\texttt{dual}}$, and hence $\bar{\mathbf{x}}^{\texttt{primal}}$ upon synchronization. In other words, given any solution $\mathbf{x} \in \{0,1\}^n$, if there exists $i \in \{1,\ldots,n\}$ for which $\bar{\mathbf{p}}_i \in \{0,1\}$ and such that $\mathbf{x}_i \neq \bar{\mathbf{p}}_i$, then we know that $f(\mathbf{x}) \geq f(\bar{\mathbf{x}}^{\texttt{primal}})$. We shall see further down how the dual process generates $\bar{\mathbf{p}}$ at any given time by identifying common elements in all the remaining unfathomed leaves of the branch-and-bound search tree. In any case, having such a partial solution allows us to restrict the search space by fixing the corresponding components of $\check{\mathbf{x}}$, and consequently this speeds up the tabu search.

Algorithm 1 summarizes our primal process. In the following section, we present our dual process.

Algorithm 1 - Primal process:

0. Initialize the tabu search with any solution $\check{\mathbf{x}} \in \{0,1\}^n$ of cost $\check{z} = f(\check{\mathbf{x}})$, and compute the corresponding transition cost vector $\boldsymbol{\delta}$. Initialize the incumbent solution $\bar{\mathbf{x}}^{\texttt{primal}} \leftarrow \check{\mathbf{x}}$, the iteration counter $\texttt{nmoves} \leftarrow 1$, the tabu status array $\boldsymbol{\tau} \leftarrow (0, 0, \ldots, 0)$, and the tabu tenure duration $\texttt{tenure} \leftarrow 10$. Initialize the counter $\texttt{elapsed} \leftarrow 0$.

1. Select w using the following index sets:

$$I^0 = \{i \in \{1, \ldots, n\} : \boldsymbol{\tau}_i < \infty\}$$
$$I^1 = \{i \in I^0 : \boldsymbol{\tau}_i < \texttt{nmoves}\}$$
$$I^2 = \{i \in I^1 : \boldsymbol{\delta}_i < 0\} \cup \{i \in I^0 : \boldsymbol{\delta}_i < f(\bar{\mathbf{x}}^{\texttt{primal}}) - f(\check{\mathbf{x}})\}$$

   (a) if I^2 is nonempty, then select w at random in $\arg\min\{\boldsymbol{\delta}_i : i \in I^2\}$,

   (b) else if I^1 is nonempty, then select w at random in I^1,

   (c) else if I^0 is nonempty, then select w at random in I^0,

   (d) else end the search because all components of $\check{\mathbf{x}}$ are fixed.

2. If $\boldsymbol{\delta}_w < 0$, then decrement $\texttt{tenure}$ by 1 if $\texttt{tenure} > 2$, else increment $\texttt{tenure}$ by 1 if $\texttt{tenure} < 10$. Update $\boldsymbol{\tau}_w \leftarrow \texttt{nmoves} + \texttt{tenure}$.

3. Perform the move by flipping the w-th component of $\check{\mathbf{x}}$. Update $\boldsymbol{\delta}$ accordingly in $O(m \log n)$ time.

4. (a) If $f(\check{\mathbf{x}}) < f(\bar{\mathbf{x}}^{\texttt{primal}})$, then update $\bar{\mathbf{x}}^{\texttt{primal}} \leftarrow \check{\mathbf{x}}$, reset $\texttt{elapsed} \leftarrow 0$, and send the new $\bar{\mathbf{x}}^{\texttt{primal}}$ to the dual process.

   (b) Increment $\texttt{nmoves}$ and $\texttt{elapsed}$ by 1. If $\texttt{elapsed}$ reaches a predefined level, then reset $\texttt{elapsed} \leftarrow 0$ and send a request to the dual process for a guiding solution.

   (c) Upon reception of a solution $\bar{\mathbf{x}}^{\texttt{dual}}$ of cost $f(\bar{\mathbf{x}}^{\texttt{dual}}) < f(\bar{\mathbf{x}}^{\texttt{primal}})$:

      i. set $\bar{\mathbf{x}}^{\texttt{primal}} \leftarrow \bar{\mathbf{x}}^{\texttt{dual}}$, set $\check{\mathbf{x}} \leftarrow \bar{\mathbf{x}}^{\texttt{dual}}$ and rebuild $\boldsymbol{\delta}$,

      ii. for all $i \in \{1, \ldots, n\}$ for which $\boldsymbol{\tau}_i < \infty$, set $\boldsymbol{\tau}_i \leftarrow 0$.

   (d) Upon reception of an improving partial solution $\bar{\mathbf{p}} \in \{0, 1, *\}^n$, for all $i \in \{1, \ldots, n\}$ for which $\bar{\mathbf{p}}_i = 0$ or 1:

      i. set $\boldsymbol{\tau}_i \leftarrow \infty$,

      ii. if $\check{\mathbf{x}}_i \neq \bar{\mathbf{p}}_i$, then flip the i-th component of $\check{\mathbf{x}}$, update $\boldsymbol{\delta}$, and increment $\texttt{nmoves}$ by 1.

   (e) Upon reception of a guiding solution $\underline{\mathbf{x}}$, for all $i \in \{1, \ldots, n\}$ for which $\boldsymbol{\tau}_i < \infty$ and $\underline{\mathbf{x}}_i = \check{\mathbf{x}}_i$, set $\boldsymbol{\tau}_i \leftarrow \texttt{nmoves} + \texttt{tenure}$.

   (f) Return to step 1.

3.3 Dual process

Our dual process computes and improves a lower bound for (P) by enumerating a lagrangian branch-and-bound search tree. In other words, the problem (P) is recursively partitioned into subproblems, and for each of these subproblems we compute a lower bound by optimizing a lagrangian dual. Recall that a lagrangian dual function maps a vector of lagrangian multipliers to the optimal value of the lagrangian relaxation parameterized by these multipliers.

This section is organized into subsections as follows. In subsection 3.3.1 we specify the subproblems into which we recursively partition (P) and we explain how we compute a lower bound for them, given any vector of multipliers. In subsection 3.3.2 we present our branch-and-bound procedure and the dual process. In subsection 3.3.4 we explain how we search for a vector of multipliers inducing a good lower bound for a given node.

3.3.1 Subproblems and lagrangian relaxation

In general, a branch-and-bound method consists in expanding a search tree by separating open leaf nodes into subnodes. This is typically done by selecting a particular 0-1 variable which is unfixed in the open node, then fixing it to 0 in one subnode and to 1 in the other. In our method we apply this to the location variables $\mathbf{x}_i$, however we also separate on the number of open locations, i.e. by limiting the value of $\sum_{i=1}^{n} \mathbf{x}_i$ within an interval $[\underline{n}, \overline{n}]$, with $1 \leq \underline{n} \leq \overline{n} \leq n$. For this reason, we shall consider subproblems of (P) defined by the following parameters:

- a partial solution vector $\mathbf{p} \in \{0, 1, *\}^n$ specifying some locations which are forced to be open or closed,
- two integers $\underline{n}$ and $\overline{n}$ specifying the minimum and maximum number of open locations.

We denote such a subproblem $SP(\mathbf{p}, \underline{n}, \overline{n})$ formulated as follows:

$$\begin{aligned}
\min \quad & \sum_{i=1}^{n} \mathbf{c}_i \mathbf{x}_i + \sum_{j=1}^{m} \sum_{i=1}^{n} \mathbf{s}_{ij} \mathbf{y}_{ij} && && (SP(\mathbf{p}, \underline{n}, \overline{n})) \\
\text{s.to} \quad & \sum_{i=1}^{n} \mathbf{y}_{ij} = 1, && 1 \leq j \leq m, && (1) \\
& 0 \leq \mathbf{y}_{ij} \leq \mathbf{x}_i, && 1 \leq i \leq n, 1 \leq j \leq m, && (2) \\
& \mathbf{x} \in \{0,1\}^n, && && (3) \\
& \mathbf{x}_i = \mathbf{p}_i, && 1 \leq i \leq n, \ \mathbf{p}_i \in \{0,1\}, && (4) \\
& \underline{n} \leq \sum_{i=1}^{n} \mathbf{x}_i \leq \overline{n}. && && (5)
\end{aligned}$$

We compute lower bounds for such subproblems using the lagrangian relaxation obtained by dualizing the m constraints (1). Denote $LR(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu})$ the lagrangian relaxation of $SP(\mathbf{p}, \underline{n}, \overline{n})$ using the vector of multipliers $\boldsymbol{\mu} \in \mathbb{R}^m$, and denote $\phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu})$ the lagrangian dual at $\boldsymbol{\mu}$:

$$\begin{aligned}
\phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = \min \ & \sum_{i=1}^{n} \mathbf{c}_i \mathbf{x}_i + \sum_{j=1}^{m} \sum_{i=1}^{n} \mathbf{s}_{ij} \mathbf{y}_{ij} + \sum_{j=1}^{m} \boldsymbol{\mu}_j \left(1 - \sum_{i=1}^{n} \mathbf{y}_{ij} \right) \\
& \text{s.to } (2-5);
\end{aligned}$$

To eliminate the decision variables $\mathbf{y}$ from the formulation, we introduce a vector of reduced costs $\bar{\mathbf{c}}^{\boldsymbol{\mu}}$ defined as follows for all $i \in \{1, \ldots, n\}$:

$$\bar{\mathbf{c}}_i^{\boldsymbol{\mu}} = \mathbf{c}_i + \sum_{j=1}^{m} \min \left\{ 0, \mathbf{s}_{ij} - \boldsymbol{\mu}_j \right\}.$$

This yields the following formulation for $LR(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu})$:

$$\phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = \sum_{j=1}^{m} \boldsymbol{\mu}_j + \min \quad \sum_{i=1}^{n} \bar{\mathbf{c}}_i^{\boldsymbol{\mu}} \mathbf{x}_i$$

$$\text{s.to} \quad \underline{n} \leq \sum_{i=1}^{n} \mathbf{x}_i \leq \overline{n}, \tag{5}$$

$$\mathbf{x}_i = \mathbf{p}_i, \qquad 1 \leq i \leq n, \ \mathbf{p}_i \in \{0,1\}, \tag{4}$$

$$\mathbf{x} \in \{0,1\}^n. \tag{3}$$

If $LR(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu})$ has no feasible solution, then for the sake of consistency we let $\phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = \infty$.

Since $\phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu})$ is a lower bound on the optimal value of $SP(\mathbf{p}, \underline{n}, \overline{n})$ for all $\boldsymbol{\mu} \in \mathbb{R}^m$, we are interested in searching for

$$\bar{\boldsymbol{\mu}} \approx \arg\max_{\boldsymbol{\mu} \in \mathbb{R}^m} \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}).$$

There exist several methods for this purpose, we shall discuss this matter further in subsection 3.3.4. We now show how to efficiently evaluate ϕ.

Evaluating a lagrangian dual function $\phi(\mathbf{p}, \underline{n}, \overline{n}, \cdot)$ for a particular vector of multipliers $\boldsymbol{\mu} \in \mathbb{R}^m$ consists in searching for an optimal solution $\underline{\mathbf{x}}$ of the corresponding lagrangian relaxation $LR(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu})$. We do this in three stages:

1. we compute the reduced costs $\bar{\mathbf{c}}^{\boldsymbol{\mu}}$;
2. we try to generate an optimal partition $\Pi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = (F^1, L^1, L^*, L^0, F^0)$ of the set of locations, i.e. $F^1 \cup L^1 \cup L^* \cup L^0 \cup F^0 = \{1, \ldots, n\}$;
3. if successful, we generate $\underline{\mathbf{x}}$ using $\Pi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu})$ and $\bar{\mathbf{c}}^{\boldsymbol{\mu}}$.

We generate the location sets F^1, L^1, L^*, L^0 and F^0 as follows. The set F^1 contains the locations fixed open, i.e. F^1 is the set of all locations $i \in \{1, \ldots, n\}$ for which $\mathbf{p}_i = 1$. Likewise, F^0 contains the locations fixed closed. Now consider constraint (5), and notice that it cannot be satisfied if $|F^1| > \overline{n}$ or if $n - |F^0| < \underline{n}$. In

this case $LR(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu})$ is infeasible, we fail to generate the partition $\Pi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu})$, and $\phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = \infty$. Otherwise, note that if $|F^1| < \underline{n}$, then in order to satisfy constraint (5) at least $\underline{n} - |F^1|$ additional unfixed locations must be open. Let L^1 be this set, the optimal decision is for L^1 to contain the unfixed locations which are the least expensive in terms of $\bar{\mathbf{c}}^{\boldsymbol{\mu}}$. Likewise, if $n - |F^0| < \overline{n}$, then at least $\overline{n} - n + |F^0|$ unfixed locations must be closed. Let L^0 be this set, the optimal decision is for L^0 to contain the unfixed locations which are the most expensive in terms of $\bar{\mathbf{c}}^{\boldsymbol{\mu}}$. Let L^* be the set of locations in $\{1, \ldots, n\}$ not in F^1, L^1, L^0 or F^0. The optimal decision for each location $i \in L^*$ is to be open or closed if $\bar{\mathbf{c}}_i^{\boldsymbol{\mu}}$ is negative or positive, respectively.

The optimal value of a lagrangian relaxation $LR(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu})$ which is feasible, i.e. if and only if $\underline{n} \leq n - |F^0|$ and $|F^1| \leq \overline{n}$, is

$$\phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = \sum_{j=1}^{m} \boldsymbol{\mu}_j + \sum_{i \in F^1 \cup L^1} \bar{\mathbf{c}}_i^{\boldsymbol{\mu}} + \sum_{i \in L^*} \min\left\{0, \bar{\mathbf{c}}_i^{\boldsymbol{\mu}}\right\}.$$

We can use the reduced costs and the optimal partition to identify an optimal solution $\underline{\mathbf{x}}$:

$$\forall i \in \{1, \ldots, n\}, \quad \underline{\mathbf{x}}_i = \begin{cases} 1 & \text{if } i \in F^1 \cup L^1, \\ 0 & \text{if } i \in L^0 \cup F^0, \\ 1 & \text{if } i \in L^* \text{ and } \bar{\mathbf{c}}_i^{\boldsymbol{\mu}} < 0, \\ 0 & \text{if } i \in L^* \text{ and } \bar{\mathbf{c}}_i^{\boldsymbol{\mu}} > 0, \\ 0 \text{ or } 1 & \text{otherwise.} \end{cases}$$

Proposition 1. *The lagrangian relaxation $LR(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu})$ can be solved in $O(mn)$ time.*

Proof. To begin with, the computation of the reduced costs $\bar{\mathbf{c}}^{\boldsymbol{\mu}}$ requires $O(mn)$ time, and the generation of the sets F^0 and F^1 requires $O(n)$ time. Next, the generation of L^1 requires us to select a certain number of the cheapest locations in

$\{1, \ldots, n\} \setminus (F^0 \cup F^1)$. This can be done using an appropriate selection algorithm such as `quickselect` which requires $O(n)$ time. We generate L^0 in a similar fashion, and generate L^* in $O(n)$ time also. Finally, we generate $\underline{\mathbf{x}}$ simply by enumerating the elements in these sets and hence the whole resolution is done in $O(mn)$ time. $\square$

We now use the concepts introduced in this subsection to present our dual process.

3.3.2 Branch-and-bound procedure and dual process

Throughout the search, the dual process maintains a incumbent solution $\bar{\mathbf{x}}^{\texttt{dual}}$ as well as a global upper bound $\bar{z}^{\texttt{dual}} = f(\bar{\mathbf{x}}^{\texttt{dual}})$. Initially, $\bar{\mathbf{x}}^{\texttt{dual}}$ is undefined and $\bar{z}^{\texttt{dual}}$ is set to ∞. The dual process then performs a preprocessing phase before solving the problem (P) by branch-and-bound. The preprocessing phase consists in computing, for all ranks $k \in \{1, \ldots, n\}$, a lower bound $\boldsymbol{\ell}_k$ on the cost of any solution of (P) with k open locations:

$$\forall \mathbf{x} \in \{0,1\}^n,\ \sum_{i=1}^{n} \mathbf{x}_i = k \implies \boldsymbol{\ell}_k \leq f(\mathbf{x}).$$

These lower bounds are subsequently used in the following manner during the resolution of (P). Note that

$$\forall \mathbf{x} \in \{0,1\}^n,\ f(\mathbf{x}) \leq \bar{z}^{\texttt{dual}} \implies \left(\sum_{i=1}^{n} \mathbf{x}_i \right) \in \left\{ k \in \{1, \ldots, n\} : \boldsymbol{\ell}_k \leq \bar{z}^{\texttt{dual}} \right\}.$$

During the resolution of (P) we therefore discard all solutions whose number of open locations is not in the interval

$$\left[\min \left\{ k \in \{1, \ldots, n\} : \boldsymbol{\ell}_k \leq \bar{z}^{\texttt{dual}} \right\}, \max \left\{ k \in \{1, \ldots, n\} : \boldsymbol{\ell}_k \leq \bar{z}^{\texttt{dual}} \right\} \right].$$

In practice, given good enough bounds $\boldsymbol{\ell}$ and $\bar{z}^{\texttt{dual}}$, this interval is small enough to justify the extra effort required for computing $\boldsymbol{\ell}$.

We begin this subsection by specifying the procedure `Branch-and-bound` which performs a lagrangian branch-and-bound search on (P), recursively subdividing the problem into subproblems of the type $SP(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu})$. We then give an overview of the dual process, in which we shall see that this procedure is applied both in the preprocessing phase to compute $\boldsymbol{\ell}$ as well as to solve (P).

3.3.2.1 Branch-and-bound

The `Branch-and-bound` procedure described here performs a lagrangian branch-and-bound search on (P), updating $\bar{\mathbf{x}}^{\texttt{dual}}$ and $\bar{z}^{\texttt{dual}}$ whenever it finds a cheaper solution. At all times, our `Branch-and-bound` procedure maintains an open node queue $\mathcal{Q}$ storing all the open leaf nodes of the search tree. Any node of the search tree is represented by a 4-tuple $(\mathbf{p}, \underline{n}, \overline{n}, \bar{\boldsymbol{\mu}})$: the subproblem corresponding to this node is $SP(\mathbf{p}, \underline{n}, \overline{n})$ and a lower bound for this node is $\phi(\mathbf{p}, \underline{n}, \overline{n}, \bar{\boldsymbol{\mu}})$. Let $*$ be the n-dimensional vector of $*$-components. We compute

$$\bar{\boldsymbol{\mu}}^{root} \approx \arg\max_{\boldsymbol{\mu} \in \mathbb{R}^m} \phi(*, 1, n, \boldsymbol{\mu})$$

using a bundle method as explained in subsection 3.3.4, and initialize the open node queue $\mathcal{Q}$ with $(*, 1, n, \bar{\boldsymbol{\mu}}^{root})$. This 4-tuple corresponds to the root node of the search tree. The `Branch-and-bound` procedure then iteratively expands the search tree until all leaf nodes have been closed, i.e. until $\mathcal{Q} = \emptyset$.

At the beginning of each iteration of `Branch-and-bound`, we select a node $(\mathbf{p}, \underline{n}, \overline{n}, \bar{\boldsymbol{\mu}}) \in \mathcal{Q}$ according to a predefined strategy `NodeSelect`. For example, our implementation applies one of the following two node selection strategies:

- `BFS`, which selects the node with the smallest lower bound in $\mathcal{Q}$ (best-first search),
- `DFS`, which selects the node most recently added into $\mathcal{Q}$ (depth-first search).

Note that since `BFS` always selects the node with the smallest lower bound, i.e. $(\mathbf{p}, \underline{n}, \overline{n}, \bar{\boldsymbol{\mu}}) \in \mathcal{Q}$ which minimizes ϕ, this lower bound is then also a global lower

bound for (P).

We then improve this lower bound by updating $\bar{\boldsymbol{\mu}}$ with a bundle method such that

$$\bar{\boldsymbol{\mu}} \approx \arg\max_{\boldsymbol{\mu}\in\mathbb{R}^m} \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}).$$

In the case where the lower bound does not exceed the upper bound $\bar{z}^{\texttt{dual}}$, we try to separate this node into subnodes by applying a given branching procedure `Branch`. We define a branching procedure `Branch` as a procedure which, when applied to a 4-tuple $(\mathbf{p}, \underline{n}, \overline{n}, \bar{\boldsymbol{\mu}})$ returns either nothing or two 3-tuples $(\mathbf{p}', \underline{n}', \overline{n}')$ and $(\mathbf{p}'', \underline{n}'', \overline{n}'')$ which satisfy the following property: for all solutions $\mathbf{x}$ feasible for $SP(\mathbf{p}, \underline{n}, \overline{n})$ and cheaper than $\bar{z}^{\texttt{dual}}$, $\mathbf{x}$ must be feasible either for $SP(\mathbf{p}', \underline{n}', \overline{n}')$ or for $SP(\mathbf{p}'', \underline{n}'', \overline{n}'')$. If $\phi(\mathbf{p}', \underline{n}', \overline{n}', \bar{\boldsymbol{\mu}}) < \bar{z}^{\texttt{dual}}$, then we insert the subnode $(\mathbf{p}', \underline{n}', \overline{n}', \bar{\boldsymbol{\mu}})$ into the open node queue $\mathcal{Q}$, and likewise for $(\mathbf{p}'', \underline{n}'', \overline{n}'', \bar{\boldsymbol{\mu}})$.

The final step of our `Branch-and-bound` iteration consists in communicating with the primal process and possibly replacing $\bar{\mathbf{x}}^{\texttt{dual}}$ to decrease $\bar{z}^{\texttt{dual}} = f(\bar{\mathbf{x}}^{\texttt{dual}})$. Let us list all types of messages which our dual process may send and receive (mirroring the list in section 3.2) before discussing how to handle them:

- Incoming:
  - receive a solution $\bar{\mathbf{x}}^{\texttt{primal}}$,
  - receive a request for a guiding solution.
- Outgoing:
  - send the solution $\bar{\mathbf{x}}^{\texttt{dual}}$,
  - send the improving partial solution $\bar{\mathbf{p}}$,
  - send the guiding solution $\underline{\mathbf{x}}$.

The first step is to try replacing $\bar{\mathbf{x}}^{\texttt{dual}}$ by a cheaper solution. Consider $\underline{\mathbf{x}}$, the optimal solution of $LR(\mathbf{p}, \underline{n}, \overline{n}, \bar{\boldsymbol{\mu}})$: if $f(\underline{\mathbf{x}}) < \bar{z}^{\texttt{dual}}$, then we replace $\bar{\mathbf{x}}^{\texttt{dual}}$ with $\underline{\mathbf{x}}$ and $\bar{z}^{\texttt{dual}}$ with $f(\underline{\mathbf{x}})$. Likewise, if we have received a solution $\bar{\mathbf{x}}^{\texttt{primal}}$ from the primal

process and if $f(\bar{\mathbf{x}}^{\texttt{primal}}) < \bar{z}^{\texttt{dual}}$, then we replace $\bar{\mathbf{x}}^{\texttt{dual}}$ with $\bar{\mathbf{x}}^{\texttt{primal}}$ and $\bar{z}^{\texttt{dual}}$ with $f(\bar{\mathbf{x}}^{\texttt{primal}})$. Note that an improved upper bound $\bar{z}^{\texttt{dual}}$ may allow us to fathom some open nodes in $\mathcal{Q}$.

Next, we occasionally update the improving partial solution $\bar{\mathbf{p}}$ and send it to the primal process if it has changed. The improving partial solution (a notion already introduced in section 3.2) is a partial solution $\bar{\mathbf{p}} \in \{0, 1, *\}^n$ which satisfies the following for all $i \in \{1, \ldots, n\}$:

$$\begin{aligned} \bar{\mathbf{p}}_i = 1 &\implies \forall(\check{\mathbf{p}}, \underline{\check{n}}, \overline{\check{n}}, \check{\boldsymbol{\mu}}) \in \mathcal{Q},\ \check{\mathbf{p}}_i = 1, \\ \bar{\mathbf{p}}_i = 0 &\implies \forall(\check{\mathbf{p}}, \underline{\check{n}}, \overline{\check{n}}, \check{\boldsymbol{\mu}}) \in \mathcal{Q},\ \check{\mathbf{p}}_i = 0. \end{aligned}$$

Since the effort required to compute an improving partial solution $\bar{\mathbf{p}}$ with a minimum number of $*$-components scales with the size of $\mathcal{Q}$, the frequency of the updates of $\bar{\mathbf{p}}$ must not be too high (this is set by a predefined parameter).

Finally, if a request for a guiding solution was received from the primal process, then we simply send $\underline{\mathbf{x}}$ as a guiding solution. If the open node queue $\mathcal{Q}$ is now empty, then we end the search, otherwise we perform the next iteration of the `Branch-and-bound` procedure.

We specify the whole `Branch-and-bound` procedure in algorithm 2. The procedure takes two callback procedures as arguments: `NodeSelect` which selects a node in the open node queue, and a branching procedure `Branch`.

Algorithm 2 - `Branch-and-bound(NodeSelect, Branch)`:

0. **Initialization -** Search for

$$\bar{\boldsymbol{\mu}}^{root} \approx \arg\max_{\boldsymbol{\mu}\in\mathbb{R}^m} \phi(*, 1, n, \boldsymbol{\mu}),$$

then initialize the open node queue $\mathcal{Q}$ with the 4-tuple $(*, 1, n, \bar{\boldsymbol{\mu}}^{root})$.

1. **Selection -** Apply `NodeSelect`$(\mathcal{Q})$ to select a 4-tuple $(\mathbf{p}, \underline{n}, \overline{n}, \bar{\boldsymbol{\mu}})$ in $\mathcal{Q}$. Remove it from $\mathcal{Q}$.

2. **Evaluation -** Update $\bar{\boldsymbol{\mu}}$ such that:

$$\bar{\boldsymbol{\mu}} \approx \arg\max_{\boldsymbol{\mu}\in\mathbb{R}^m} \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}).$$

3. **Separation -**

   (a) If $\phi(\mathbf{p}, \underline{n}, \overline{n}, \bar{\boldsymbol{\mu}}) \geq \bar{z}^{\texttt{dual}}$, then go straight to step 4.

   (b) Apply `Branch`$(\mathbf{p}, \underline{n}, \overline{n}, \bar{\boldsymbol{\mu}})$ to try to generate two 3-tuples $(\mathbf{p}', \underline{n}', \overline{n}')$ and $(\mathbf{p}'', \underline{n}'', \overline{n}'')$.

   (c) If successful and if $\phi(\mathbf{p}', \underline{n}', \overline{n}', \bar{\boldsymbol{\mu}}) < \bar{z}^{\texttt{dual}}$, then insert $(\mathbf{p}', \underline{n}', \overline{n}')$ into $\mathcal{Q}$. Do the same for $(\mathbf{p}'', \underline{n}'', \overline{n}'')$.

4. **Upper bound update and communication -**

   (a) Let $\underline{\mathbf{x}}$ be the optimal solution of $LR(\mathbf{p}, \underline{n}, \overline{n}, \bar{\boldsymbol{\mu}})$. If we have received a new solution $\bar{\mathbf{x}}^{\texttt{primal}}$ from the primal process, then let $\mathbf{x} = \arg\min\{f(\underline{\mathbf{x}}), f(\bar{\mathbf{x}}^{\texttt{primal}})\}$, else let $\mathbf{x} = \underline{\mathbf{x}}$. If $f(\mathbf{x}) < \bar{z}^{\texttt{dual}}$, then:

      i. Set $\bar{\mathbf{x}}^{\texttt{dual}} \leftarrow \mathbf{x}$ and $\bar{z}^{\texttt{dual}} \leftarrow f(\mathbf{x})$.

      ii. If $\bar{\mathbf{x}}^{\texttt{dual}} = \underline{\mathbf{x}}$ then send the new best solution $\bar{\mathbf{x}}^{\texttt{dual}}$ to the primal process.

      iii. For all $(\check{\mathbf{p}}, \underline{\check{n}}, \overline{\check{n}}, \check{\boldsymbol{\mu}}) \in \mathcal{Q}$: if $\phi(\check{\mathbf{p}}, \underline{\check{n}}, \overline{\check{n}}, \check{\boldsymbol{\mu}}) \geq \bar{z}^{\texttt{dual}}$ then remove $(\check{\mathbf{p}}, \underline{\check{n}}, \overline{\check{n}}, \check{\boldsymbol{\mu}})$ from $\mathcal{Q}$.

   (b) Occasionally, do the following:

      i. Initialize $\bar{\mathbf{p}} \leftarrow *$.

      ii. For all $i \in \{1, \ldots, n\}$: if for all $(\check{\mathbf{p}}, \underline{\check{n}}, \overline{\check{n}}, \check{\boldsymbol{\mu}}) \in \mathcal{Q}$ we have $\check{\mathbf{p}}_i = 0$, then set $\bar{\mathbf{p}}_i \leftarrow 0$,

      iii. For all $i \in \{1, \ldots, n\}$: if for all $(\check{\mathbf{p}}, \underline{\check{n}}, \overline{\check{n}}, \check{\boldsymbol{\mu}}) \in \mathcal{Q}$ we have $\check{\mathbf{p}}_i = 1$, then set $\bar{\mathbf{p}}_i \leftarrow 1$.

      iv. Send the improving partial solution $\bar{\mathbf{p}}$ to the primal process.

   (c) Upon reception of a request for a guiding solution, send $\underline{\mathbf{x}}$ to the primal process as a guiding solution.

   (d) If $\mathcal{Q}$ is nonempty, then go back to step 1.

3.3.2.2 Dual process overview

At the beginning of this subsection we mentioned that the dual process begins by performing a preprocessing phase to generate a vector $\boldsymbol{\ell} \in \mathbb{R}^n$ which satisfies

$$\forall \mathbf{x} \in \{0,1\}^n,\ \sum_{i=1}^{n} \mathbf{x}_i = k \implies \boldsymbol{\ell}_k \leq f(\mathbf{x}).$$

For this purpose, we initialize $\boldsymbol{\ell}_k \leftarrow \infty$ for all $k \in \{1, \ldots, n\}$, and apply `Branch-and-bound` using the branching procedure `BrP` specified in algorithm 3. This procedure branches on the number of open locations $\sum_{i=1}^{n} \mathbf{x}_i$. We thus separate (P) into subproblems $SP(*, \underline{n}, \overline{n})$ with $1 \leq \underline{n} \leq \overline{n} \leq n$.

Algorithm 3 - `BrP`$(\mathbf{p}, \underline{n}, \overline{n}, \bar{\boldsymbol{\mu}})$:

---

case $\underline{n} = \overline{n}$

set $\boldsymbol{\ell}_{\underline{n}} \leftarrow \phi(\mathbf{p}, \underline{n}, \overline{n}, \bar{\boldsymbol{\mu}})$ and return $\emptyset$;

case $\phi(\mathbf{p}, \underline{n}, \underline{n}, \bar{\boldsymbol{\mu}}) \geq \bar{z}^{\texttt{dual}}$

return `BrP`$(\mathbf{p}, \underline{n} + 1, \overline{n}, \bar{\boldsymbol{\mu}})$;

case $\phi(\mathbf{p}, \overline{n}, \overline{n}, \bar{\boldsymbol{\mu}}) \geq \bar{z}^{\texttt{dual}}$

return `BrP`$(\mathbf{p}, \underline{n}, \overline{n} - 1, \bar{\boldsymbol{\mu}})$;

case otherwise

return $(\mathbf{p}, \underline{n}, \lfloor \frac{\underline{n}+\overline{n}}{2} \rfloor)$ and $(\mathbf{p}, \lfloor \frac{\underline{n}+\overline{n}}{2} \rfloor + 1, \overline{n})$.

Note that this branching procedure is recursive, and that it requires repeated evaluations of ϕ. In general this is done in $O(mn)$ time, however we shall see in subsection 3.3.3 that it is possible to compute $\Pi(\mathbf{p}, \underline{n}+1, \overline{n}, \bar{\boldsymbol{\mu}})$ and $\phi(\mathbf{p}, \underline{n}+1, \overline{n}, \bar{\boldsymbol{\mu}})$ in $O(n)$ time by reusing $\bar{\mathbf{c}}^{\bar{\boldsymbol{\mu}}}$, $\Pi(\mathbf{p}, \underline{n}, \overline{n}, \bar{\boldsymbol{\mu}})$ and $\phi(\mathbf{p}, \underline{n}, \overline{n}, \bar{\boldsymbol{\mu}})$. The same can also be done for $(\mathbf{p}, \underline{n}, \overline{n} - 1, \bar{\boldsymbol{\mu}})$. When `Branch-and-bound` terminates, we set $\boldsymbol{\ell}_k \leftarrow \bar{z}^{\texttt{dual}}$ for all $k \in \{1, \ldots, n\}$ for which $\boldsymbol{\ell}_k = \infty$. In this manner, for all $k \in \{1, \ldots, n\}$, $\boldsymbol{\ell}_k$ is a lower bound on the solutions of (P) with k open locations, i.e. on the solutions

of $SP(*, k, k)$.

Having completed preprocessing, we solve (P) by applying `Branch-and-bound` using the branching procedure `BrS` specified in algorithm 4. For all $\mathbf{p} \in \{0, 1, *\}^n$ and for all $k \in \{1, \ldots, n\}$, we define $\mathbf{p}^{+k}$ and $\mathbf{p}^{-k}$ as follows. For all $i \in \{1, \ldots, n\}$,

$$\mathbf{p}_i^{+k} = \begin{cases} 1 & \text{if } k = i, \\ \mathbf{p}_i & \text{otherwise.} \end{cases} \qquad \mathbf{p}_i^{-k} = \begin{cases} 0 & \text{if } k = i, \\ \mathbf{p}_i & \text{otherwise.} \end{cases}.$$

Algorithm 4 - $\texttt{BrS}(\mathbf{p}, \underline{n}, \overline{n}, \bar{\boldsymbol{\mu}})$:

---

case $\mathbf{p} \in \{0, 1\}^n$

return $\emptyset$,

case $\boldsymbol{\ell}_{\underline{n}} \geq \bar{z}^{\texttt{dual}}$ **or** $\phi(\mathbf{p}, \underline{n}, \underline{n}, \bar{\boldsymbol{\mu}}) \geq \bar{z}^{\texttt{dual}}$

return $\texttt{BrS}(\mathbf{p}, \underline{n} + 1, \overline{n}, \bar{\boldsymbol{\mu}})$;

case $\boldsymbol{\ell}_{\overline{n}} \geq \bar{z}^{\texttt{dual}}$ **or** $\phi(\mathbf{p}, \overline{n}, \overline{n}, \bar{\boldsymbol{\mu}}) \geq \bar{z}^{\texttt{dual}}$

return $\texttt{BrS}(\mathbf{p}, \underline{n}, \overline{n} - 1, \bar{\boldsymbol{\mu}})$;

case $\exists k \in \{1, \ldots, n\},\ p_k = *,\ \phi(\mathbf{p}^{+k}, \underline{n}, \overline{n}, \bar{\boldsymbol{\mu}}) \geq \bar{z}^{\texttt{dual}}$

return $\texttt{BrS}(\mathbf{p}^{-k}, \underline{n}, \overline{n}, \bar{\boldsymbol{\mu}})$;

case $\exists k \in \{1, \ldots, n\},\ p_k = *,\ \phi(\mathbf{p}^{-k}, \underline{n}, \overline{n}, \bar{\boldsymbol{\mu}}) \geq \bar{z}^{\texttt{dual}}$

return $\texttt{BrS}(\mathbf{p}^{+k}, \underline{n}, \overline{n}, \bar{\boldsymbol{\mu}})$;

case otherwise

select $k = \arg\min \left\{ |\bar{\mathbf{c}}_i^{\bar{\boldsymbol{\mu}}}| : p_i = *, i \in \{1, \ldots, n\} \right\}$,

return $(\mathbf{p}^{+k}, \underline{n}, \overline{n})$ and $(\mathbf{p}^{-k}, \underline{n}, \overline{n})$.

In a sense, for all $i \in \{1, \ldots, n\}$ such that $p_i = *$, the value $|\bar{\mathbf{c}}_i^{\bar{\boldsymbol{\mu}}}|$ is a measure of the impact of the decision to branch on the unfixed location i, and `BrS` chooses to branch on the unfixed location with the smallest impact. Like `BrS`, this procedure

is recursive and seemingly requires many $O(mn)$ evaluations of ϕ which in fact can be performed in $O(n)$. We explain how in the following subsection.

3.3.3 Reoptimization

Given any vector $\boldsymbol{\mu} \in \mathbb{R}^m$, suppose that we have evaluated $\phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu})$ and that this value is not ∞. We thus also have the reduced costs $\bar{\mathbf{c}}^{\boldsymbol{\mu}}$ as well as an optimal partition $\Pi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = (F^1, L^1, L^*, L^0, F^0)$. The following proposition and its corollary specify how to reuse these to compute $\phi(\mathbf{p}, \underline{n}+k, \overline{n}, \boldsymbol{\mu})$ and $\phi(\mathbf{p}, \underline{n}, \overline{n}-k, \boldsymbol{\mu})$ in $O(n)$ time, for all $k > 0$. Without loss of generality, suppose that the values in $\bar{\mathbf{c}}^{\boldsymbol{\mu}}$ are all different.

Proposition 2. *For all $k > 0$, if $\underline{n} + k > \min\{\overline{n}, n - |F^0|\}$ then*

$$\phi(\mathbf{p}, \underline{n} + k, \overline{n}, \boldsymbol{\mu}) = \infty,$$

else if $\underline{n} + k \leq |F^1|$ then

$$\begin{aligned}\Pi(\mathbf{p}, \underline{n} + k, \overline{n}, \boldsymbol{\mu}) &= (F^1, \emptyset, L^*, L^0, F^0),\\ \phi(\mathbf{p}, \underline{n} + k, \overline{n}, \boldsymbol{\mu}) &= \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}),\end{aligned}$$

else, let $\tilde{L}$ be the set of the k least expensive locations in L^* in terms of $\bar{\mathbf{c}}^{\boldsymbol{\mu}}$, we have

$$\begin{aligned}\Pi(\mathbf{p}, \underline{n} + k, \overline{n}, \boldsymbol{\mu}) &= (F^1, L^1 \cup \tilde{L}, L^* \setminus \tilde{L}, L^0, F^0),\\ \phi(\mathbf{p}, \underline{n} + k, \overline{n}, \boldsymbol{\mu}) &= \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) + \sum_{i \in \tilde{L}} \max\left\{0, \bar{\mathbf{c}}_i^{\boldsymbol{\mu}}\right\}.\end{aligned}$$

Proof. If $\underline{n} + k > \overline{n}$, then by definition of $SP(\mathbf{p}, \underline{n} + k, \overline{n})$, this subproblem is infeasible, hence $\phi(\mathbf{p}, \underline{n} + k, \overline{n}, \boldsymbol{\mu}) = \infty$. Likewise, if $\underline{n} + k > n - |F^0|$, then there exists no $\mathbf{x} \in \{0, 1\}^n$ which can satisfy both $\underline{n} + k \leq \sum_{i=1}^n \mathbf{x}_i$ and $\mathbf{x}_i = 0$ for all $i \in \{1, \ldots, n\}$ for which $\mathbf{p}_i = 0$, i.e. for all $i \in F^0$.

Suppose henceforth that $\underline{n} + k \leq \min\{\overline{n}, n - |F^0|\}$, and let $\Pi(\mathbf{p}, \underline{n} + k, \overline{n}, \boldsymbol{\mu}) =$

$(F'^1, L'^1, L'^*, L'^0, F'^0)$. We know that $F'^1 = F^1$ and $F'^0 = F^0$ because both these sets are induced solely by $\mathbf{p}$.

If $\underline{n} + k \leq |F^1|$, then all $\mathbf{x} \in \{0,1\}^n$ which satisfy $\mathbf{x}_i = 1$ for all $i \in F^1$ also satisfy $\underline{n} + k \leq \sum_{i=1}^n \mathbf{x}_i$, consequently L^1 and L'^1 are empty by definition of these sets. Likewise, $L'^0 = L^0$ and $L'^* = L^*$, hence $\Pi(\mathbf{p}, \underline{n} + k, \overline{n}, \boldsymbol{\mu}) = \Pi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu})$ and $\phi(\mathbf{p}, \underline{n} + k, \overline{n}, \boldsymbol{\mu}) = \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu})$.

Suppose now that $\underline{n} + k > |F^1|$. The set L'^1 consists of the $\underline{n} + k - |F^1|$ unfixed locations which are least expensive in terms of $\bar{\mathbf{c}}^{\boldsymbol{\mu}}$, and we know that this is a subset of $L^1 \cup L^*$ because

$$\min\{\overline{n}, n - |F^0|\} = n - |L^0 \cup F^0| = |F^1| + |L^1| + |L^*|,$$

hence $L'^1 = L^1 \cup \tilde{L}$ and $L'^* = L^* \setminus \tilde{L}$. Consequently,

$$\begin{aligned}
\phi(\mathbf{p}, \underline{n} + k, \overline{n}, \boldsymbol{\mu}) &= \sum_{i \in F'^1 \cup L'^1} \bar{\mathbf{c}}_i^{\boldsymbol{\mu}} + \sum_{i \in L'^*} \min\left\{0, \bar{\mathbf{c}}_i^{\boldsymbol{\mu}}\right\}, \\
&= \sum_{i \in F^1 \cup L^1} \bar{\mathbf{c}}_i^{\boldsymbol{\mu}} + \sum_{i \in \tilde{L}} \bar{\mathbf{c}}_i^{\boldsymbol{\mu}} + \sum_{i \in L^*} \min\left\{0, \bar{\mathbf{c}}_i^{\boldsymbol{\mu}}\right\} - \sum_{i \in \tilde{L}} \min\left\{0, \bar{\mathbf{c}}_i^{\boldsymbol{\mu}}\right\}, \\
&= \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) + \sum_{i \in \tilde{L}} \max\left\{0, \bar{\mathbf{c}}_i^{\boldsymbol{\mu}}\right\}.
\end{aligned}$$

□

Corollary 1. *For all $k > 0$, if $\overline{n} - k < \max\{\underline{n}, |F^1|\}$ then*

$$\phi(\mathbf{p}, \underline{n}, \overline{n} - k, \boldsymbol{\mu}) = \infty,$$

else if $n - (\overline{n} - k) \leq |F^0|$ then

$$\begin{aligned}
\Pi(\mathbf{p}, \underline{n}, \overline{n} - k, \boldsymbol{\mu}) &= (F^1, L^1, L^*, \emptyset, F^0), \\
\phi(\mathbf{p}, \underline{n}, \overline{n} - k, \boldsymbol{\mu}) &= \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}),
\end{aligned}$$

else, let $\tilde{L}$ *be the set of the* k *most expensive locations in* L^* *in terms of* $\bar{\mathbf{c}}^{\boldsymbol{\mu}}$*, we have*

$$\Pi(\mathbf{p}, \underline{n}, \overline{n} - k, \boldsymbol{\mu}) = (F^1, L^1, L^* \setminus \tilde{L}, L^0 \cup \tilde{L}, F^0),$$
$$\phi(\mathbf{p}, \underline{n}, \overline{n} - k, \boldsymbol{\mu}) = \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) - \sum_{i \in \tilde{L}} \min\left\{0, \bar{\mathbf{c}}_i^{\boldsymbol{\mu}}\right\}.$$

The next propositions state how to reoptimize $LR(\mathbf{p}^{+k}, \underline{n}, \overline{n}, \boldsymbol{\mu})$ for all $k \in \{1, \ldots, n\}$ for which $\mathbf{p}_k = *$, i.e. for all $k \in (L^1 \cup L^* \cup L^0)$. For this purpose we need to identify the following locations:

$$i^1_> = \arg\max\left\{\bar{\mathbf{c}}_i^{\boldsymbol{\mu}} : i \in L^1\right\}, \qquad i^*_> = \arg\max\left\{\bar{\mathbf{c}}_i^{\boldsymbol{\mu}} : i \in L^*\right\},$$
$$i^*_< = \arg\min\left\{\bar{\mathbf{c}}_i^{\boldsymbol{\mu}} : i \in L^*\right\}, \qquad i^0_< = \arg\min\left\{\bar{\mathbf{c}}_i^{\boldsymbol{\mu}} : i \in L^0\right\}.$$

Each proposition also has a corollary stating a symmetric result for $\mathbf{p}^{-k}$.

Proposition 3. *For all* $k \in L^1$*,*

$$\Pi(\mathbf{p}^{+k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = (F^1 \cup \{k\}, L^1 \setminus \{k\}, L^*, L^0, F^0),$$
$$\phi(\mathbf{p}^{+k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}).$$

Proof. Let $\Pi(\mathbf{p}^{+k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = (F'^1, L'^1, L'^*, L'^0, F'^0)$. Since $\mathbf{p}^{+k}$ differs from $\mathbf{p}$ only in its k-th component, for which we have $p_k = *$ and $\mathbf{p}^{+k} = 1$, it follows that $F'^1 = F^1 \cup \{k\}$ and that $F'^0 = F^0$. Recall that by definition, if $|F'^1| \geq \underline{n}$ then $L'^1 = \emptyset$ else L'^1 contains the $\underline{n} - |F'^1|$ cheapest unfixed locations in terms of $\bar{\mathbf{c}}^{\boldsymbol{\mu}}$. Consequently, we have $L'^1 = L^1 \setminus \{k\}$, and by using a similar reasoning we also have $L'^0 = L^0$, hence $L'^* = L^*$. Note that $F'^1 \cup L'^1 = F^1 \cup L^1$, hence $\phi(\mathbf{p}^{+k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu})$ □

Corollary 2. *For all $k \in L^0$,*

$$\Pi(\mathbf{p}^{-k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = (F^1, L^1, L^*, L^0 \setminus \{k\}, F^0 \cup \{k\}),$$
$$\phi(\mathbf{p}^{-k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}).$$

Proposition 4. *For all $k \in L^*$, if L^1 is empty then*

$$\Pi(\mathbf{p}^{+k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = (F^1 \cup \{k\}, \emptyset, L^* \setminus \{k\}, L^0, F^0),$$
$$\phi(\mathbf{p}^{+k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) + \max\left\{0, \bar{\mathbf{c}}_k^{\boldsymbol{\mu}}\right\},$$

else

$$\Pi(\mathbf{p}^{+k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = (F^1 \cup \{k\}, L^1 \setminus \{i_>^1\}, (L^* \setminus \{k\}) \cup \{i_>^1\}, L^0, F^0),$$
$$\phi(\mathbf{p}^{+k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) + \max\left\{0, \bar{\mathbf{c}}_k^{\boldsymbol{\mu}}\right\} - \max\left\{0, \bar{\mathbf{c}}_{i_>^1}^{\boldsymbol{\mu}}\right\}.$$

Proof. Let $\Pi(\mathbf{p}^{+k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = (F'^1, L'^1, L'^*, L'^0, F'^0)$. As in the proof of proposition 3, we have $F'^1 = F^1 \cup \{k\}$, $L'^0 = L^0$, and $F'^0 = F^0$. If L^1 is empty, then $|F^1| \geq \underline{n}$ and therefore L'^1 is empty also, hence:

$$\begin{aligned}
\phi(\mathbf{p}^{+k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) &= \sum_{i \in F'^1 \cup L'^1} \bar{\mathbf{c}}_i^{\boldsymbol{\mu}} + \sum_{i \in L'^*} \min\left\{0, \bar{\mathbf{c}}_i^{\boldsymbol{\mu}}\right\}, \\
&= \sum_{i \in F^1 \cup L^1} \bar{\mathbf{c}}_i^{\boldsymbol{\mu}} + \sum_{i \in \tilde{L}} \bar{\mathbf{c}}_i^{\boldsymbol{\mu}} + \sum_{i \in L^*} \min\left\{0, \bar{\mathbf{c}}_i^{\boldsymbol{\mu}}\right\} - \sum_{i \in \tilde{L}} \min\left\{0, \bar{\mathbf{c}}_i^{\boldsymbol{\mu}}\right\}, \\
&= \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) + \sum_{i \in \tilde{L}} \max\left\{0, \bar{\mathbf{c}}_i^{\boldsymbol{\mu}}\right\}.
\end{aligned}$$

If L^1 is not empty, then L'^1 must contains one location less than L^1, which therefore

must be $i^1_>$. Consequently, $i^1_> \in L'^*$ and

$$\begin{aligned}
\phi(\mathbf{p}^{+k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) &= \sum_{i \in F^1 \cup \{k\} \cup L^1 \setminus \{i^1_>\}} \bar{\mathbf{c}}^{\boldsymbol{\mu}}_i + \sum_{i \in (L^* \setminus \{k\}) \cup \{i^1_>\}} \min\left\{0, \bar{\mathbf{c}}^{\boldsymbol{\mu}}_i\right\}, \\
&= \sum_{i \in F^1 \cup L^1} \bar{\mathbf{c}}^{\boldsymbol{\mu}}_i + \bar{\mathbf{c}}^{\boldsymbol{\mu}}_k - \bar{\mathbf{c}}^{\boldsymbol{\mu}}_{i^1_>} \\
&\qquad + \sum_{i \in L^*} \min\left\{0, \bar{\mathbf{c}}^{\boldsymbol{\mu}}_i\right\} - \min\left\{0, \bar{\mathbf{c}}^{\boldsymbol{\mu}}_k\right\} + \min\left\{0, \bar{\mathbf{c}}^{\boldsymbol{\mu}}_{i^1_>}\right\}, \\
&= \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) + \max\left\{0, \bar{\mathbf{c}}^{\boldsymbol{\mu}}_k\right\} - \max\left\{0, \bar{\mathbf{c}}^{\boldsymbol{\mu}}_{i^1_>}\right\}.
\end{aligned}$$

□

Corollary 3. *For all $k \in L^*$, if L^0 is empty then*

$$\begin{aligned}
\Pi(\mathbf{p}^{-k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) &= (F^1, L^1, L^* \setminus \{k\}, \emptyset, F^0 \cup \{k\}), \\
\phi(\mathbf{p}^{-k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) &= \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) - \min\left\{0, \bar{\mathbf{c}}^{\boldsymbol{\mu}}_k\right\},
\end{aligned}$$

else

$$\begin{aligned}
\Pi(\mathbf{p}^{-k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) &= (F^1, L^1, (L^* \setminus \{k\}) \cup \{i^0_<\}, L^0 \setminus \{i^0_<\}, F^0 \cup \{k\}), \\
\phi(\mathbf{p}^{-k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) &= \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) - \min\left\{0, \bar{\mathbf{c}}^{\boldsymbol{\mu}}_k\right\} + \min\left\{0, \bar{\mathbf{c}}^{\boldsymbol{\mu}}_{i^0_<}\right\}.
\end{aligned}$$

Proposition 5. *For all $k \in L^0$, if L^1 and L^* are both empty then*

$$\phi(\mathbf{p}^{+k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = \infty,$$

else if L^1 is nonempty and L^* is empty then

$$\begin{aligned}
\Pi(\mathbf{p}^{+k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) &= (F^1 \cup \{k\}, L^1 \setminus \{i^1_>\}, \emptyset, (L^0 \setminus \{k\}) \cup \{i^1_>\}, F^0), \\
\phi(\mathbf{p}^{+k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) &= \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) + \bar{\mathbf{c}}^{\boldsymbol{\mu}}_k - \bar{\mathbf{c}}^{\boldsymbol{\mu}}_{i^1_>},
\end{aligned}$$

else if L^1 *is empty and* L^* *is nonempty then*

$$\Pi(\mathbf{p}^{+k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = (F^1 \cup \{k\}, \emptyset, L^* \setminus \{i^*_>\}, (L^0 \setminus \{k\}) \cup \{i^*_>\}, F^0),$$
$$\phi(\mathbf{p}^{+k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) + \bar{\mathbf{c}}^{\boldsymbol{\mu}}_k - \min\left\{0, \bar{\mathbf{c}}^{\boldsymbol{\mu}}_{i^*_>}\right\},$$

else

$$\Pi(\mathbf{p}^{+k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = (F^1 \cup \{k\}, L^1 \setminus \{i^1_>\}, (L^* \setminus \{i^*_>\}) \cup \{i^1_>\}, (L^0 \setminus \{k\}) \cup \{i^*_>\}, F^0),$$
$$\phi(\mathbf{p}^{+k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) + \bar{\mathbf{c}}^{\boldsymbol{\mu}}_k - \max\left\{0, \bar{\mathbf{c}}^{\boldsymbol{\mu}}_{i^1_>}\right\} - \min\left\{0, \bar{\mathbf{c}}^{\boldsymbol{\mu}}_{i^*_>}\right\}.$$

Proof. If $L^1 = L^* = \emptyset$, then $|\{i \in \{1, \ldots, n\} : \mathbf{p}_i = 1\}| = |F^1| = \overline{n}$ and thus $SP(\mathbf{p}^{+k}, \underline{n}, \overline{n})$ is infeasible and $\phi(\mathbf{p}^{+k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = \infty$. Suppose henceforth that $L^1 \cup L^* \neq \emptyset$, and let $\Pi(\mathbf{p}^{+k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = (F'^1, L'^1, L'^*, L'^0, F'^0)$. Again, as in the proof of proposition 3, we have $F'^1 = F^1 \cup \{k\}$ and $F'^0 = F^0$. Also, $k \in L^0$ implies that L^0 is not empty and hence $|F^0| + |L^0| = n - \overline{n}$.

In the case where $L^1 \neq \emptyset$ and $L^* = \emptyset$, we have $|F^1| + |L^1| = \underline{n} = \overline{n}$. Consequently L'^1 must contain one location less than L^1, this location therefore being $i^1_>$, and L'^0 must remain the same size as L^0, hence $L'^0 = (L^0 \setminus \{k\}) \cup \{i^1_>\}$.

In the case where $L^1 = \emptyset$ and $L^* = \emptyset$, we have $|F^1| \geq \underline{n}$ and thus L'^1 is also empty. Consequently L'^0 must remain the same size as L^0, hence $L'^0 = (L^0 \setminus \{k\}) \cup \{i^*_>\}$.

In the final case where both L^1 and L^* are nonempty, we have $|F^1| + |L^1| = \underline{n}$. Similarly as in both previous cases, L'^1 must contain one location less than L^1 and hence $L^1 = L^1 \setminus \{i^*_>\}$, and L'^0 must remain the same size as L^0. Note that $\bar{\mathbf{c}}^{\boldsymbol{\mu}}_{i^*_>} = \max_{i \in L^*} \bar{\mathbf{c}}^{\boldsymbol{\mu}}_i < \bar{\mathbf{c}}^{\boldsymbol{\mu}}_{i^1_>}$, hence $L'^0 = (L^0 \setminus \{k\}) \cup \{i^*_>\}$. We deduce the values of $\phi(\mathbf{p}^{+k}, \underline{n}, \overline{n}, \boldsymbol{\mu})$ like in the proof of proposition 4. □

Corollary 4. *For all* $k \in L^1$*, if* L^* *and* L^0 *are both empty then*

$$\phi(\mathbf{p}^{-k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = \infty,$$

else if L^* is empty and L^0 is nonempty then

$$\Pi(\mathbf{p}^{-k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = (F^1, (L^1 \setminus \{k\}) \cup \{i^0_<\}, \emptyset, L^0 \setminus \{i^0_<\}, F^0 \cup \{k\}),$$
$$\phi(\mathbf{p}^{-k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) - \bar{\mathbf{c}}^{\boldsymbol{\mu}}_i + \bar{\mathbf{c}}^{\boldsymbol{\mu}}_{i^0_<},$$

else if L^* is nonempty and L^0 is empty then

$$\Pi(\mathbf{p}^{-k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = (F^1, (L^1 \setminus \{k\}) \cup \{i^*_<\}, L^* \setminus \{i^*_<\}, \emptyset, F^0 \cup \{k\}),$$
$$\phi(\mathbf{p}^{-k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) - \bar{\mathbf{c}}^{\boldsymbol{\mu}}_i + \max\left\{0, \bar{\mathbf{c}}^{\boldsymbol{\mu}}_{i^*_<}\right\},$$

else

$$\Pi(\mathbf{p}^{-k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = (F^1, (L^1 \setminus \{k\}) \cup \{i^*_<\}, (L^* \setminus \{i^*_<\}) \cup \{i^0_<\}, L^0 \setminus \{i^0_<\}, F^0 \cup \{k\}),$$
$$\phi(\mathbf{p}^{-k}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) - \bar{\mathbf{c}}^{\boldsymbol{\mu}}_i + \min\left\{0, \bar{\mathbf{c}}^{\boldsymbol{\mu}}_{i^0_<}\right\} + \max\left\{0, \bar{\mathbf{c}}^{\boldsymbol{\mu}}_{i^*_<}\right\}.$$

3.3.4 Bundle Method

In this subsection we describe how we adapt a bundle method to optimize the lagrangian dual of any subproblem $SP(\mathbf{p}, \underline{n}, \overline{n})$ with $\mathbf{p} \in \{0, 1, *\}^n$, $1 \leq \underline{n} \leq \overline{n} \leq n$:

$$\bar{\boldsymbol{\mu}} \approx \arg\max_{\boldsymbol{\mu} \in \mathbb{R}^m} \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}),$$

Recall that such subproblems correspond to nodes evaluated by the `Branch-and-bound` procedure, and that the optimization of the lagrangian duals takes place in steps 0 and 2 of algorithm 2.

There exist many different methods to compute $\bar{\boldsymbol{\mu}}$: the simplex method, dual-ascent heuristics [15], the volume algorithm [3], simple subgradient search, Nesterov's algorithm [39], etc. However, our preliminary experiments have indicated that bundle methods works well for this problem. Indeed, on the one hand, optimizing $\phi(\mathbf{p}, \underline{n}, \overline{n}, \cdot)$ exactly with the simplex method is too difficult in practice for all but the easiest and smallest instances of the UFLP, and in any case we are

not interested in exact optimization per se. On the other hand, subgradient search methods might not capture enough information about $\phi(\mathbf{p}, \underline{n}, \overline{n}, \cdot)$ during the optimization process to be as effective as the bundle method, as suggested by some preliminary experiments with a bundle method using very small bundle size limits. Bundle methods therefore seem like a good compromise between two extremes. After giving a brief summary of the bundle method, we shall explain how we adapted it to solve our problem. A detailed description of bundle and trust-region methods in general lies beyond the scope of this paper, see, e.g., [28] for a detailed treatment on the subject.

3.3.4.1 Overview

The bundle method is a trust-region method which optimizes a lagrangian dual function iteratively until a termination criterion is met. At each iteration t, a *trial point* $\boldsymbol{\mu}^{(t)} \in \mathbb{R}^m$ is determined. In the case of the first iteration (i.e. $t = 0$), the trial point depends on where we are in the dual process:

- during preprocessing, in step 0 of algorithm 2: this is the very first application of the bundle method, and the initial trial point is the result of a dual-ascent heuristic such as `DualLoc`;
- during the resolution of (P), in step 0 of algorithm 2: the initial trial point is $\bar{\boldsymbol{\mu}}^{\texttt{root}}$, the best vector of multipliers for the root node (computed during preprocessing);
- otherwise, i.e. in step 2 of alg. 2: the initial trial point is $\bar{\boldsymbol{\mu}}$, the best vector of multipliers for the parent node.

At any subsequent iteration $t > 0$, $\boldsymbol{\mu}^{(t)}$ is determined by optimizing a *model* approximating $\phi(\mathbf{p}, \underline{n}, \overline{n}, \cdot)$.

Next, $\phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}^{(t)})$ is evaluated, and this value and a subgradient are used to update the model of the dual. A new iteration is then performed, unless either a

convergence criterion, or a predefined iteration limit, or a bound limit is reached, in which case we update the best multipliers:

$$\bar{\boldsymbol{\mu}} \leftarrow \arg\max_{t} \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}^{(t)}).$$

The convergence criterion is satisfied when the bundle method determines that its model of the dual $\phi(\mathbf{p}, \underline{n}, \overline{n}, \cdot)$ provides a good enough approximation of the dual in the neighborhood of an optimal vector of multipliers. The bound limit is reached when the current trial point $\boldsymbol{\mu}^{(t)}$ satisfies $\phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}^{(t)}) \geq \bar{z}$, where $\bar{z}$ is an upper bound on the optimal value of (P).

The model of the lagrangian dual is specified using a bundle $\mathcal{B}$, defined by a set of vector-scalar pairs $(\boldsymbol{\sigma}, \epsilon)$ such that the vector $\boldsymbol{\sigma} \in \mathbb{R}^m$ and the scalar $\epsilon \in \mathbb{R}$ define a first-order outer approximation of $\phi(\mathbf{p}, \underline{n}, \overline{n}, \cdot)$, i.e.

$$\forall \boldsymbol{\mu} \in \mathbb{R}^m, \quad \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) \leq \sum_{j=1}^{m} \boldsymbol{\sigma}_j \boldsymbol{\mu}_j + \epsilon.$$

At each iteration t, the bundle $\mathcal{B}$ (and hence, the model of $\phi(\mathbf{p}, \underline{n}, \overline{n}, \cdot)$) is updated using a pair $(\boldsymbol{\sigma}^{(t)}, \epsilon^{(t)})$ generated as follows. First, we determine an optimal solution $\underline{\mathbf{x}}^{(t)}$ of the lagrangian relaxation $LR(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}^{(t)})$. Assuming that it exists, let

$$z^{(t)} = \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}^{(t)}) = \sum_{j=1}^{m} \boldsymbol{\mu}_j^{(t)} + \sum_{i=1}^{n} \bar{\mathbf{c}}_i^{\boldsymbol{\mu}^{(t)}} \underline{\mathbf{x}}_i^{(t)}.$$

To determine a subgradient $\boldsymbol{\sigma}^{(t)} \in \mathbb{R}^m$, we specify a vector $\underline{\mathbf{y}}^{(t)}$ associated with the optimal solution $\underline{\mathbf{x}}^{(t)}$ of $LR(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}^{(t)})$:

$$\forall i \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, m\}, \quad \underline{\mathbf{y}}_{ij}^{(t)} = \begin{cases} 0 & \text{if } \underline{\mathbf{x}}_i^{(t)} = 0 \text{ or } \mathbf{s}_{ij} - \boldsymbol{\mu}_j^{(t)} > 0, \\ 1 & \text{if } \underline{\mathbf{x}}_i^{(t)} = 1 \text{ and } \mathbf{s}_{ij} - \boldsymbol{\mu}_j^{(t)} < 0, \\ 0 \text{ or } 1 & \text{otherwise.} \end{cases}$$

For all suitable $\underline{\mathbf{y}}^{(t)}$, the vector in $\mathbb{R}^m$ with components $\left(1 - \sum_{i=1}^{n} \underline{\mathbf{y}}_{ij}^{(t)}\right)_j$ for all $j \in \{1, \ldots, m\}$ is a subgradient of $\phi(\mathbf{p}, \underline{n}, \overline{n}, \cdot\,)$. We let

$$\boldsymbol{\sigma}_j^{(t)} = 1 - \left|\left\{i \in \{1, \ldots, n\} : \underline{\mathbf{x}}_i^{(t)} = 1, \mathbf{s}_{ij} < \boldsymbol{\mu}_j^{(t)}\right\}\right|, \quad \forall j \in \{1, \ldots, m\}.$$

and

$$\epsilon^{(t)} = z^{(t)} - \sum_{j=1}^{m} \boldsymbol{\sigma}_j^{(t)} \boldsymbol{\mu}_j^{(t)}.$$

The function $\phi(\mathbf{p}, \underline{n}, \overline{n}, \cdot\,)$ is concave, thus $\phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) \leq \sum_{j=1}^{m} \boldsymbol{\sigma}_j^{(t)} \boldsymbol{\mu}_j + \epsilon^{(t)}$ for all $\boldsymbol{\mu} \in \mathbb{R}^m$. We therefore update the bundle $\mathcal{B}$ using the pair $(\boldsymbol{\sigma}^{(t)}, \epsilon^{(t)})$ specifying a first-order outer approximation of $\phi(\mathbf{p}, \underline{n}, \overline{n}, \cdot\,)$ at $\boldsymbol{\mu}^{(t)}$.

3.3.4.2 Subgradient cache

Recall that the subproblems of the form $SP(\mathbf{p}, \underline{n}, \overline{n})$ are associated with the nodes in a branch-and-bound search tree, therefore if a bundle method is to be performed at each node, then it stands to reason that some lagrangian dual functions may be very similar to one another. In particular, one $(\boldsymbol{\sigma}, \epsilon)$ pair generated during the optimization of one dual function may also specify a valid first-order outer approximation function of another dual function. For this reason, our method maintains a cache $\mathcal{C}$ in which $(\boldsymbol{\sigma}, \epsilon)$ pairs are stored as soon as they are generated. Thus, before optimizing any dual function $\phi(\mathbf{p}, \underline{n}, \overline{n}, \cdot\,)$, our method retrieves some suitable pairs from $\mathcal{C}$ to populate the current bundle (which is otherwise empty), thus improving the model and hopefully also the quality of the trial points. Note that although there exist other ways of re-using the subgradient information generated during the application of the bundle method, ours is simple, effective, and requires only a limited amount of memory. We now introduce the result which tells us if and how a pair is suitable.

Proposition 6. *Suppose that* $\underline{\mathbf{x}}$ *is an optimal solution of any lagrangian relaxation* $LR(\mathbf{p}', \underline{n}', \overline{n}', \boldsymbol{\mu})$*, and let* $(\boldsymbol{\sigma}, \epsilon)$ *be the corresponding pair which defines a first-order outer approximation of* $\phi(\mathbf{p}', \underline{n}', \overline{n}', \cdot\,)$*. This pair defines a first-order outer approx-*

imation of any other function $\phi(\mathbf{p}'', \underline{n}'', \overline{n}'', \cdot)$ *if* $\underline{\mathbf{x}}$ *is also an optimal solution of* $LR(\mathbf{p}'', \underline{n}'', \overline{n}'', \boldsymbol{\mu})$.

Proof. Since $\boldsymbol{\sigma}$ is generated using only $\bar{\mathbf{c}}^{\boldsymbol{\mu}}$ and $\underline{\mathbf{x}}$, and since $\underline{\mathbf{x}}$ is optimal for $LR(\mathbf{p}'', \underline{n}'', \overline{n}'', \boldsymbol{\mu})$ as well as for $LR(\mathbf{p}', \underline{n}', \overline{n}', \boldsymbol{\mu})$, then:

- $\phi\left(\mathbf{p}'', \underline{n}'', \overline{n}'', \boldsymbol{\mu}\right) = \phi\left(\mathbf{p}', \underline{n}', \overline{n}', \boldsymbol{\mu}\right) = \sum_{j=1}^{m} \boldsymbol{\mu}_j + \sum_{i=1}^{n} \bar{\mathbf{c}}_i^{\boldsymbol{\mu}}\, \underline{\mathbf{x}}_i$,
- $\boldsymbol{\sigma}$ is a subgradient of $\phi(\mathbf{p}'', \underline{n}'', \overline{n}'', \cdot)$ as well as of $\phi(\mathbf{p}', \underline{n}', \overline{n}', \cdot)$.

This last function is concave, therefore for all $\tilde{\boldsymbol{\mu}} \in \mathbb{R}^m$:

$$\phi\left(\mathbf{p}'', \underline{n}'', \overline{n}'', \tilde{\boldsymbol{\mu}}\right) \leq \sum_{j=1}^{m} \boldsymbol{\sigma}_j \tilde{\boldsymbol{\mu}}_j + \left(\phi\left(\mathbf{p}'', \underline{n}'', \overline{n}'', \boldsymbol{\mu}\right) - \sum_{j=1}^{m} \boldsymbol{\sigma}_j \boldsymbol{\mu}_j\right),$$

$$\phi\left(\mathbf{p}'', \underline{n}'', \overline{n}'', \tilde{\boldsymbol{\mu}}\right) \leq \sum_{j=1}^{m} \boldsymbol{\sigma}_j \tilde{\boldsymbol{\mu}}_j + \epsilon.$$

□

Remark 1. Suppose that $\mathbf{p}''$, $\underline{n}''$ and $\overline{n}''$ are such that $SP(\mathbf{p}'', \underline{n}'', \overline{n}'')$ is a subproblem of $SP(\mathbf{p}', \underline{n}', \overline{n}')$. In this special case, $\underline{\mathbf{x}}$ only needs to be feasible for $SP(\mathbf{p}'', \underline{n}'', \overline{n}'')$, because it is then also optimal for $LR(\mathbf{p}'', \underline{n}'', \overline{n}'', \boldsymbol{\mu})$.

We therefore specify the cache $\mathcal{C}$ as containing key-value pairs of the form $((\underline{\mathbf{x}}, \mathbf{p}', \underline{n}', \overline{n}'), (\boldsymbol{\sigma}, \epsilon))$. Prior to optimizing a dual function $\phi(\mathbf{p}'', \underline{n}'', \overline{n}'', \cdot)$, we update the bundle using $(\boldsymbol{\sigma}, \epsilon)$ pairs obtained by traversing $\mathcal{C}$ for keys which match the following conditions:

- $\underline{n}' \leq \underline{n}'' \leq \sum_{i=1}^{n} \underline{\mathbf{x}}_i \leq \overline{n}'' \leq \overline{n}'$, and
- for all $i \in \{1, \ldots, n\}$: either $\underline{\mathbf{x}}_i = \mathbf{p}_i''$ or $\mathbf{p}_i'' = *$, and either $\mathbf{p}_i'' = \mathbf{p}_i'$ or $\mathbf{p}_i' = *$.

When properly implemented, these tests can be performed quickly enough for the cache to be of practical use.

Consider now the problem of populating such a cache $\mathcal{C}$. The straightforward way to do so is as follows: at each iteration t of the bundle method applied to any

dual function $\phi(\mathbf{p}, \underline{n}, \overline{n}, \cdot)$, simply add the key-value pair $\big((\underline{\mathbf{x}}^{(t)}, \mathbf{p}, \underline{n}, \overline{n}),\ (\boldsymbol{\sigma}^{(t)}, \epsilon^{(t)})\big)$. However, we can instead try to improve this key in terms of increasing the likelihood of subsequently reusing $(\boldsymbol{\sigma}^{(t)}, \epsilon^{(t)})$. For this purpose we generate a vector $\tilde{\mathbf{p}}$ in the following manner, using the partition $\Pi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}^{(t)}) = (F^1, L^1, L^*, L^0, F^0)$ as well as the reduced costs $\bar{\mathbf{c}}^{\boldsymbol{\mu}^{(t)}}$ computed during the evaluation of $\phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}^{(t)})$. Let

$$\tilde{F}^1 = \left\{ i \in F^1 : \forall k \in (L^* \cup L^0),\ \bar{\mathbf{c}}_i^{\boldsymbol{\mu}^{(t)}} \leq \bar{\mathbf{c}}_k^{\boldsymbol{\mu}^{(t)}} \right\},$$
$$\tilde{F}^0 = \left\{ i \in F^1 : \forall k \in (L^1 \cup L^*),\ \bar{\mathbf{c}}_i^{\boldsymbol{\mu}^{(t)}} \geq \bar{\mathbf{c}}_k^{\boldsymbol{\mu}^{(t)}} \right\},$$

we initialize $\tilde{\mathbf{p}} \leftarrow \mathbf{p}$ and set $\tilde{\mathbf{p}}_i \leftarrow *$ for all $i \in \tilde{F}^1 \cup \tilde{F}^0$.

Proposition 7. *This partial solution* $\tilde{\mathbf{p}}$ *is such that* $\underline{\mathbf{x}}^{(t)}$ *is also an optimal solution of* $LR(\tilde{\mathbf{p}}, \underline{n}, \overline{n}, \boldsymbol{\mu}^{(t)})$.

Proof. The locations in $\tilde{F}^1$ form a subset of F^1 and are not more expensive (in terms of $\bar{\mathbf{c}}^{\boldsymbol{\mu}^{(t)}}$) than any location in L^* or L^0, therefore there exists an optimal partition $\Pi(\tilde{\mathbf{p}}, \underline{n}, \overline{n}, \boldsymbol{\mu}^{(t)}) = (F'^1, L'^1, L'^*, L'^0, F'^0)$ such that $\tilde{F}^1 \subset L'^1$. Likewise, there also exists L'^0 such that $\tilde{F}^0 \subset L'^0$. Consequently, $F'^1 \cup L'^1 = F^1 \cup L^1$, $F'^0 \cup L'^0 = F^0 \cup L^0$ and $L'^* = L^*$. □

Corollary 5. $\phi(\tilde{\mathbf{p}}, \underline{n}, \overline{n}, \boldsymbol{\mu}^{(t)}) = \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}^{(t)})$.

This result ensures that $(\boldsymbol{\sigma}^{(t)}, \epsilon^{(t)})$ is also a first-order outer approximation of $\phi(\tilde{\mathbf{p}}, \underline{n}, \overline{n}, \cdot)$ at $\boldsymbol{\mu}^{(t)}$. By associating the key $(\underline{\mathbf{x}}^{(t)}, \tilde{\mathbf{p}}, \underline{n}, \overline{n})$ instead of $(\underline{\mathbf{x}}^{(t)}, \mathbf{p}, \underline{n}, \overline{n})$ with the pair $(\boldsymbol{\sigma}^{(t)}, \epsilon^{(t)})$ in $\mathcal{C}$, we increase its potential for reuse.

A proper implementation of such a cache is crucial to ensure good performance. Details are provided in the appendix.

3.4 Computational Results

We begin this section by outlining how our implementation executes the primal and dual processes concurrently. Following this, we present our test parameters and methodology, and then we present our results. The complete source code, in C, can be freely obtained from the authors.

3.4.1 Concurrency

Recall that the primal and dual processes both consist of a loop: in the primal process one iteration corresponds to one 1OPT move, and in the dual process one iteration corresponds to one branch-and-bound node. Although it could be, our implementation is not parallel. Instead it performs iterations of the primal and dual processes sequentially and alternately. The dual and primal iterations are not performed in equal proportions: at the beginning of the search 100 primal iterations are performed for every dual iteration, and this ratio progressively inverts itself until 100 dual iterations are performed for every primal iteration. The inversion speed is set by a predefined parameter, and in our implementation the default setting is such that a few thousand primal iterations should have been performed before the steady state of 100 dual per 1 primal iterations is reached. This policy seems to work well for reducing overall CPU time for all instances. Indeed the primal process is much more likely to find an improving solution at the beginning of the search than later on, and vice-versa for the dual process. Additionally, the dual process also benefits from having a good upper bound early in the search.

3.4.2 Parameters

The machines which we used for our tests are identical workstations equipped with Intel i7-2600 CPUs clocked at 3.4 GHz. We have tested our implementation on all instances in `UflLib`, with a maximum execution time limit set at 2 hours of CPU time. To date, the `UflLib` collection consists of the following instance sets: `Bilde-Krarup`; `Chessboard`; `Euclidian`; `Finite projective planes` (`k=11`, `k=17`); `Galvão-Raggi`; `Körkel-Ghosh` (symmetric and asymmetric, $n = m \in \{250, 500, 750\}$); `Barahona-Chudak`; `Gap` (`a`, `b`, `c`); `M*`; `Beasley` (a.k.a. `ORLIB`); `Perfect codes`; `Uniform`. Most of these instances are such that $n = m = 100$, but in the largest case (some instances in `Barahona-Chudak`) we have $n = m = 3000$. Part of the motivation behind applying our method on so many instances stems from how few results on the exact resolution of the UFLP

have been published in the last decade.

We performed some preliminary experiments to determine good maximal bundle size settings, and we noticed that execution times seem to be little affected by this parameter. Without going into the details, it appears that for comparatively small ($n = m = 100$) and easy `UflLib` instances, a maximal bundle size between 20 or 50 is best. However since we hope to solve the larger and more difficult instances, we set this parameter to 100 for all of our tests. We use the following bundle method iteration limits: `iter_limit_initial` = 2000 for the very first application of the bundle method during the dual process; `iter_limit_other` = 120 for all the subsequent applications. Other search parameters are set as follows: the improving partial solution $\bar{p}$ is updated after every 250 consecutive iterations of the dual process, or whenever the best-known solution is updated; likewise, the primal process sends requests for guiding solutions after 250 consecutive primal process iterations with no improvement of the best-known solution. Again, these settings are not very sensitive.

3.4.3 Tests

Our first series of tests aims to illustrate the effectiveness of our branching rules and of maintaining a cache $\mathcal{C}$. We performed these tests on the 30 instances in `Gap-a`, which are the easier instances in `Gap`. The instances in this set are such that $n = m = 100$, and they are characterized by their large duality gap, inducing suitably large search trees for our tests to be meaningful. Because we are not measuring anything related to the primal process, in our tests we disable the primal process and initialize the upper bound $\bar{z}^{\texttt{dual}}$ with the optimal value of the corresponding instance. As a consequence, when solving any one instance, the search trees obtained at the end of the dual process should in theory be identical to one another whichever node selection strategy is used (`BFS` or `DFS`). In practice, many components of our implementation are not numerically stable and hence results differ a little. In fact the mean difference in the number of nodes of the search trees obtained by solving with `DFS` and with `BFS` is 5.4%, with a standard

deviation of 3.3%.

Figure 3.1 illustrates the profile curves corresponding to solving with `DFS` and the following settings:

'`-cache`' without cache,

'`+cache`' with cache, limited to 1024 elements,

'`-branch`' without preprocessing, and only considering the first and last cases in the branching procedures `BrP` and `BrS` (i.e. cases '$\underline{n} = \overline{n}$', '$\mathbf{p} \in \{0,1\}^n$' and 'otherwise'),

'`+branch`' with preprocessing, with `BrP` and `BrS` as specified in algorithms 3 and 4.

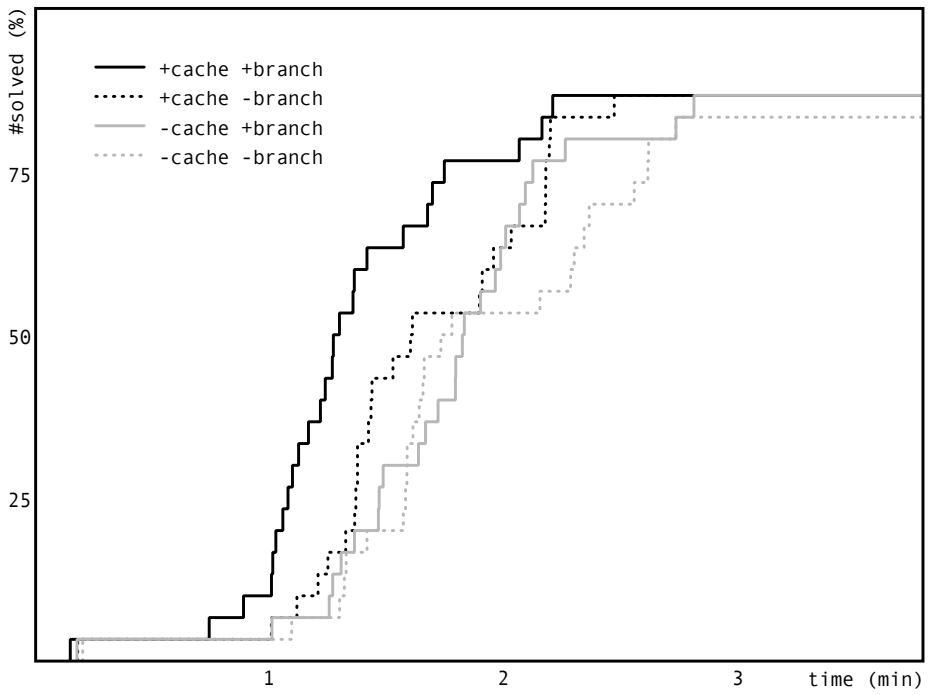
Observe that '`+branch`' is always worthwhile when used in conjunction with '`+cache`', despite the extra computational effort involved. In fact, performance without the cache is still improved when solving the more difficult instances. For this reason, we use '`+branch`' in all of our subsequent tests.

Figure 3.2 illustrates the profile curves corresponding to solving with `DFS` (unless otherwise specified) with the associated maximal cache size. For example, the curve '`1024`' corresponds to the results using `DFS` and $|\mathcal{C}| < 1024$. Notice that the cache does not improve performance when used in conjunction with `BFS`, unlike with `DFS`. This is not surprising because in large branch-and-bound trees, the nodes consecutively selected by `BFS` often bear little relation to one another and too few elements of the cache are reused for it to be worthwhile. Conversely, the nodes consecutively selected by `DFS` are often closely related. Otherwise, we can see that the use of a cache in conjunction with `DFS` improves performance provided that the cache remains small enough.

3.4.4 Solving UflLib

Finally, we apply our method on all instances in `UflLib` with the following two settings:

Figure 3.1: Preprocessing and branching settings.

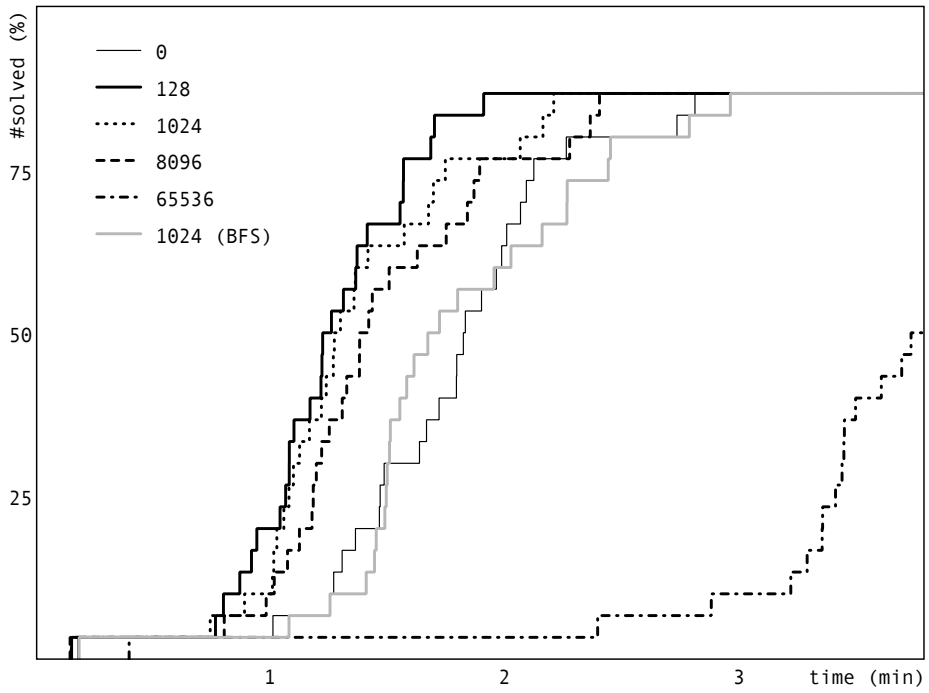


1. `DFS` and a cache with less than 128 elements,

2. `BFS` and no cache.

Tables 3.I and 3.II summarize the respective execution time results for each instance set. The columns 'min', 'median', 'max' list the execution time of the easiest, median and hardest instance in each set, respectively. The column 'mean' lists the mean execution time for the instances in each set which were solved before reaching the time limit, and the column 'std.dev.' lists the corresponding standard deviation. Timeouts are represented by a dash (recall that the time limit is 2 CPU-hours), all other times are in minutes and seconds, rounded up. Figure 3.3 illustrates the corresponding profile curves for the instance sets whose hardest instance required at least one minute: `Barahona-Chudak`, `Finite projective planes`, `Gap`, `Körkel-Ghosh` and `M*`.

Figure 3.2: Cache settings.



First, note that the only instance set for which our method does not work well is `Barahona-Chudak`. These are large instances, for which

$$n = m \in \{500, 1000, 1500, 2000, 2500, 3000\}.$$

We noticed that the tabu search does not always work very well, and that our branch-and-bound approach often fails to partition (P) adequately. Conversely, the semi-lagrangian approach of Beltran-Royo et al. [7] was able to solve 16 out of these 18 instances to optimality in a matter of seconds. Likewise, the primal-dual approach of Hansen et al. [26] also works very well on instances of this type.

Nonetheless, our method works well for all other instances in `UflLib`. We can see that several instance sets are better solved by one setting than by the other. This is especially the case for the `Finite projective planes` instances, for which '`DFS` with $|\mathcal{C}| < 128$' performs very badly, in stark contrast to '`BFS`'. Looking more closely we noticed that these poor results were correlated to a very poor incumbent

Table 3.I: Solving with DFS and $|\mathcal{C}| < 128$.

instances	solved	min	median	max	mean	std.dev.
Barahona-Chudak	6 / 18	1"	-	-	15'07"	25'42"
Beasley	17 / 17	1"	1"	1"	1"	1"
Bilde-Krarup	220 / 220	1"	1"	3"	1"	1"
Chessboard	30 / 30	1"	1"	12"	1"	3"
Euclidian	30 / 30	1"	1"	1"	1"	1"
FPP k=11	30 / 30	1"	7"	3'59"	35"	1'05"
FPP k=17	28 / 30	2"	10'22"	-	22'02"	30'11"
Gap-a	30 / 30	31"	5'05"	20'09"	7'27"	5'39"
Gap-b	30 / 30	3'13"	11'49"	35'02"	12'43"	7'29"
Gap-c	30 / 30	70'13"	83'60"	117'15"	86'38"	9'25"
Galvao-Raggi	50 / 50	1"	1"	1"	1"	1"
KG-250	27 / 30	3"	12'28"	-	24'39"	35'45"
KG-500	7 / 30	44'31"	-	-	70'08"	22'10"
KG-750	0 / 30	-	-	-	-	-
M*	22 / 22	1"	1"	40'50"	1'57"	8'30"
Perfect Codes	32 / 32	1"	1"	1"	1"	1"
Uniform	30 / 30	1"	3"	17"	3"	4"

solution. Therefore it follows that the tabu search in the primal process remains stuck in a local optimality trap very early in the search. It is therefore up to the dual process to send information to the primal process to enable it to escape from its trap. Evidently this does occur with BFS node selection, but not with DFS. We performed another series of tests on the Finite projective planes instances using BFS node selection but with the primal process disabled. We found that the dual process was able to find the optimal solution on its own in a matter of seconds, but not as quickly as in the original 'BFS' tests. This illustrates the pertinence of the cooperative aspect of our approach.

Note that in the case of the Gap instance set, the easier instances in the a and b subsets are better solved with 'BFS' while the opposite is true of the harder instances in the c subset. On the other hand, the 'DFS with $|\mathcal{C}| < 128$' setting performs better for M* and Körkel-Ghosh. The instances in these sets have not yet all been solved to optimality, unlike all other sets in UflLib. Until now, only about half of those in M* had known optima, yet our method solves all instances

Table 3.II: Solving with `BFS` and no cache.

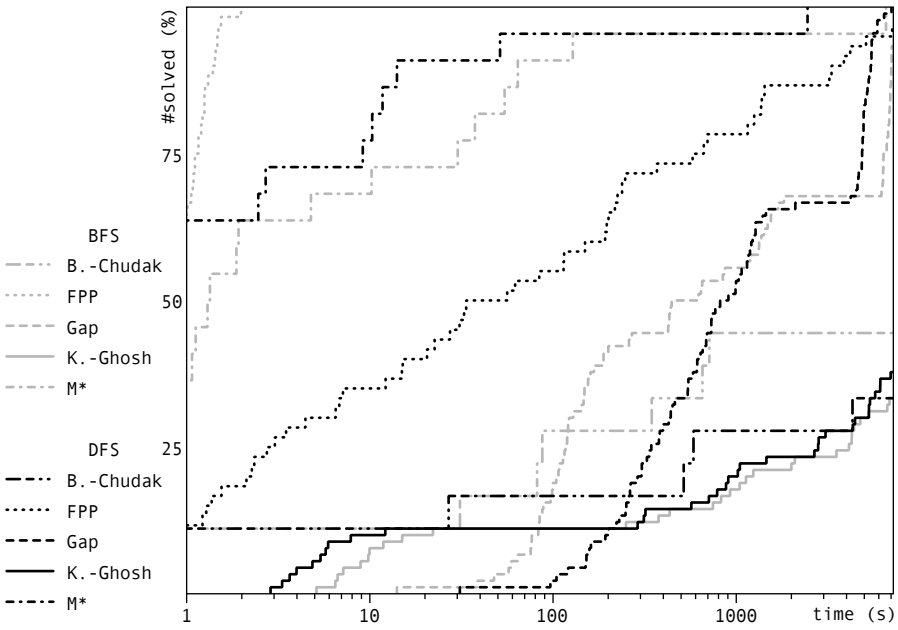
instances	solved	min	median	max	mean	std.dev.
Barahona-Chudak	8 / 18	1"	-	-	3'60"	4'37"
Beasley	17 / 17	1"	1"	1"	1"	1"
Bilde-Krarup	220 / 220	1"	1"	3"	1"	1"
Chessboard	30 / 30	1"	1"	1"	1"	1"
Euclidian	30 / 30	1"	1"	1"	1"	1"
FPP k=11	30 / 30	1"	1"	1"	1"	1"
FPP k=17	30 / 30	1"	2"	2"	2"	1"
Gap-a	30 / 30	15"	1'57"	25'46"	4'01"	6'35"
Gap-b	30 / 30	39"	7'10"	30'22"	10'24"	9'20"
Gap-c	24 / 30	7'24"	115'08"	-	107'39"	21'26"
Galvao-Raggi	50 / 50	1"	1"	1"	1"	1"
KG-250	27 / 30	6"	14'36"	-	25'52"	34'40"
KG-500	3 / 30	71'07"	-	-	74'31"	3'25"
KG-750	0 / 30	-	-	-	-	-
M*	22 / 22	1"	2"	109'29"	5'15"	22'46"
Perfect Codes	32 / 32	1"	1"	1"	1"	1"
Uniform	30 / 30	1"	4"	23"	5"	5"

but the largest one to optimality in less that a minute on a modern computer, and the largest one in less than an hour. Our method solves all instances in `M*` and a good proportion of those in `Körkel-Ghosh` to optimality. Also, to the best of our knowledge, only 2 of the 90 instances in `Körkel-Ghosh` had known optima [7], yet our method was able to find 28 more. We present these results in detail in the appendix.

3.5 Concluding remarks

In this paper, we presented a cooperative method to solve (P) exactly, in which a primal process and a dual process exchange information to improve the search. The primal process performs a variation of the tabu search method proposed by Michel et al. [35], and the dual process performs a lagrangian branch-and-bound search. We selected this particular metaheuristic for its simplicity and its good overall performance, however any other metaheuristic can be used instead.

Figure 3.3: Profile curves for `UflLib`.



Our main contributions lie in the dual process. Partitionning (P) into subproblems $SP(\mathbf{p}, \underline{n}, \overline{n})$ allows us to apply sophisticated branching strategies. Our branching rules rely on results for quickly reoptimizing lagrangian relaxations with modified parameters $\mathbf{p}$, $\underline{n}$ or $\overline{n}$, and hence require relatively little computational effort. Note that the branch-and-bound method presented in this paper could be supplemented with the preprocessing and branching rules presented by Goldengorin et al. [21, 22].

Furthermore, we introduced a subgradient caching technique which helps improve performance of the bundle method, which we apply to compute a lower bound at each node. The subgradients are stored in the cache as soon as they are generated during the application of the bundle method, and may be retrieved to initialize the bundle at the beginning of a subsequent application. We introduced results to improve the potential for reuse of a subgradient within the specific context of the UFLP, but these possibly could be extended to lagrangian branch-and-bound methods for other problems. Maintaining a subgradient cache may not be the ideal way of reusing information obtained during the optimization of the lagrangian dual,

but this technique has certain practical advantages: it requires constant time and space, it is conceptually simple (and hence, easy to implement), its behavior is easy to control and it works well enough in conjunction with a depth-first node selection strategy.

Note also that by computing the lower bound at each node using a bundle method allows us a certain level of control on the computational effort expended at each node, which more or less directly translates into bound quality. This is in contrast to the dual-ascent heuristic of `DualLoc`, and also to solvers such as CPLEX which apply the dual simplex algorithm.

Finally, we presented extensive computational results. Our method performs well for a wide variety of problem instances, several of which having been solved to optimality for the first time.

Acknowledgements

We wish to thank Paul-Virak Khuong who kindly helped us with our code, improving its performance significantly. We are also grateful to Antonio Frangioni for his bundle method implementation and for taking the time to answer our questions.

3.6 Appendices

3.6.1 Bundle method and cache implementation details

3.6.1.1 Cache

An efficient implementation of $\mathcal{C}$ is crucial to its performance. In our implementation, $\mathcal{C}$ is a FIFO queue whose maximal size is a predefined parameter. Consequently, an insertion of a new element into a full $\mathcal{C}$ is preceded by the removal of its oldest element. When inserting a new element into $\mathcal{C}$, our implementation does not examine the contents of $\mathcal{C}$ to detect if it is already present, but instead it tests its presence using a counting Bloom filter. A Bloom filter is a data structure originally proposed by Bloom [9] which allows testing efficiently whether a set contains an

element (with occasional false positives, but no false negatives). A counting Bloom filter is a variant defined using any vector $\boldsymbol{\beta}$ of positive integers and a set H of different hash functions $h \in H$ which each map any element possibly in $\mathcal{C}$ to an index of $\boldsymbol{\beta}$:

- if an element e is added into $\mathcal{C}$, then increment $\boldsymbol{\beta}_{h(e)}$ by 1 for all $h \in H$,
- therefore an element e is certainly not in $\mathcal{C}$ if there exists $h \in H$ for which $\boldsymbol{\beta}_{h(e)} = 0$,
- if an element e is removed from $\mathcal{C}$, then decrement $\boldsymbol{\beta}_{h(e)}$ by 1 for all $h \in H$.

Bloom filters are efficient in time and in space even for very large sets. By hard-coding the maximum size of $\mathcal{C}$ to $2^{16} - 1$ elements, we implemented $\boldsymbol{\beta}$ as an array of 16-bit unsigned integers. In our implementation, the dimension of this array is in the low hundred thousands, and $|H| = 7$. As a consequence, the probability of false positives when testing whether $e \in \mathcal{C}$ is less than 1%.

When trying to insert a new element e into $\mathcal{C}$, the following steps are performed:

1. If $\boldsymbol{\beta}_{h(e)} > 0$ for all $h \in H$, then e is probably already in $\mathcal{C}$: go to step 4.
2. If $\mathcal{C}$ is full, then remove the oldest element e' in $\mathcal{C}$ and decrement $\boldsymbol{\beta}_{h(e')}$ by 1 for all $h \in H$.
3. Insert e into $\mathcal{C}$ and increment $\boldsymbol{\beta}_{h(e)}$ by 1 for all $h \in H$.
4. Done.

The counting Bloom filter thus ensures the unicity of the elements in $\mathcal{C}$.

3.6.1.2 Bundle method iteration limits

Recall that the elements stored in $\mathcal{C}$ are key-value pairs specifying first-order outer approximations of lagrangian dual functions, and the purpose of $\mathcal{C}$ is to provide valid approximations to populate the bundle. Recall also that in our method we apply a bundle method during the evaluation of each node $(\mathbf{p}, \underline{n}, \overline{n}, \bar{\boldsymbol{\mu}})$ selected

in $\mathcal{Q}$ by the `Branch-and-bound` procedure, and that the initial trial point $\boldsymbol{\mu}^{(0)}$ is set to $\bar{\boldsymbol{\mu}}$, i.e. the best vector of multipliers found for the parent node. Once our implementation has identified a subset S of valid $(\boldsymbol{\sigma}, \epsilon)$ pairs present in $\mathcal{C}$, it does not immediately update the bundle with all pairs in S. We have found it more convenient to update the bundle using at most b pairs in S, with b being the maximum bundle size minus 1. This seemingly arbitrary choice was partly due to certain limitations of `[B]TT` (the bundle method implementation which we use), however it seems to be effective. In the case where the valid subset size exceeds this number, we update the bundle with the b pairs in S selected as follows: let $\boldsymbol{\mu}^{(0)}$ be the initial trial point of the bundle method, we select $(\boldsymbol{\sigma}, \epsilon) \in S$ minimizing the value $\sum_{j=1}^{m} \boldsymbol{\mu}^{(0)} \boldsymbol{\sigma}_j + \epsilon$. In this manner, we select the most accurate first-order outer approximations, the value $\sum_{j=1}^{m} \boldsymbol{\mu}^{(0)} \boldsymbol{\sigma}_j + \epsilon - \phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}^{(0)})$ being the error at $\boldsymbol{\mu}^{(0)}$ of the approximation defined by $(\boldsymbol{\sigma}, \epsilon)$.

Recall that in our implementation, a hard limit on the number of iterations to be performed by an application of the bundle method is set. This is either `iter_limit_initial` for the very first application (at the beginning of which the cache is empty), or `iter_limit_other` for the others. In all applications other than the first, we use `iter_limit_other` $- \min\{b, |S|\}$ as the actual iteration limit. In this manner, and provided that `iter_limit_other` $> b$, the intensity of the search for the optimal multiplier is inversely proportional to the size of the initial bundle, and hence indirectly to the quality of the model of the lagrangian dual.

3.6.1.3 Fast floating point arithmetic

A large part of the computational effort is expended in the following computations within the bundle method:

- computing reduced costs $\bar{\mathbf{c}}^{\boldsymbol{\mu}}$,
- generating a subgradient $\boldsymbol{\sigma}^{(t)}$,
- computing $\sum_{j=1}^{m} \boldsymbol{\mu}_j^{(0)} \boldsymbol{\sigma}_j + \epsilon$.

While the theoretical complexity of these operations cannot be reduced, in practice we can speed these up by performing as many of them as possible in the SSE registers which are now standard on x86 processors. Consider the computation of the reduced costs given $\boldsymbol{\mu} \in \mathbb{R}^m$:

$$\forall i \in \{1, \ldots, n\}, \quad \bar{\mathbf{c}}_i^{\boldsymbol{\mu}} \leftarrow \mathbf{c}_i + \sum_{j=1}^{m} \min\left\{0, \mathbf{s}_{ij} - \boldsymbol{\mu}_j\right\}.$$

The sum of the minima can be performed in the SSE registers several minima at a time. Similarly, the computation of $\sum_{j=1}^{m} \boldsymbol{\mu}_j^{(0)} \boldsymbol{\sigma}_j + \epsilon$ for all suitable $(\boldsymbol{\sigma}, \epsilon)$ pairs identified at the beginning of a bundle method consists mainly of a dot product. This operation can be performed by the BLAS function `ddot`, and recent implementations of BLAS will perform several products simultaneously.

The generation of the subgradient $\boldsymbol{\sigma}^{(t)}$ was specified earlier as follows:

$$\forall j \in \{1, \ldots, m\}, \quad \boldsymbol{\sigma}_j^{(t)} \leftarrow 1 - \left|\left\{i \in \{1, \ldots, n\} : \underline{\mathbf{x}}_i^{(t)} = 1, \mathbf{s}_{ij} < \boldsymbol{\mu}_j^{(t)}\right\}\right|,$$

but can be rewritten using the following vectors $\mathbf{v}^i \in \{0,1\}^m$, $i \in \{1, \ldots, n\}$:

$$\boldsymbol{\sigma}^{(t)} \leftarrow (1, 1, \ldots, 1) - \sum_{\substack{i=1 \\ \underline{\mathbf{x}}_i^{(t)}=1}}^{n} \mathbf{v}^i,$$

$$\text{with } \forall i \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, m\}, \ \mathbf{v}_j^i = \begin{cases} 1 & \text{if } \mathbf{s}_{ij} < \boldsymbol{\mu}_j^{(t)}, \\ 0 & \text{otherwise.} \end{cases}$$

Using SSE registers, we can compute several components of a vector $\mathbf{v}^i$ simultaneously.

3.6.2 Computational results

Table 3.III illustrates our results for the instances of the `Körkel-Ghosh` set which we were able to solve to optimality. To the best of our knowledge, most of

these were unknown until now. Table 3.IV illustrates our results for the `M*` set, which we were able to solve to optimality in its entirety in less than an hour. The column '#nodes' lists the number of nodes in the search tree for each instance, and the column '#lag.eval.' lists the number of times a lagrangian relaxation was solved as per subsection 3.3.1. The column '#moves' lists the number of 1OPT moves performed by the primal process. As before, all times are rounded up.

Table 3.III: Optimal values for the Körkel-Ghosh instance set

instance	opt	time	#nodes	#lag.eval.	#moves
ga250a-2	257502	5'20"	69,950	4,352,212	10,829
ga250a-4	257987	116'04"	1,555,510	96,300,841	96,301
ga250b-1	276296	24'21"	452,946	22,557,355	23,118
ga250b-2	275141	4'50"	87,894	4,467,390	10,092
ga250b-3	276093	13'07"	249,354	12,146,379	16,655
ga250b-4	276332	17'27"	324,970	15,686,289	18,916
ga250b-5	276404	15'05"	286,054	13,942,841	17,883
ga250c-1	334135	6"	2,510	80,553	966
ga250c-2	330728	3"	1,220	39,043	639
ga250c-3	333662	5"	2,106	67,897	878
ga250c-4	332423	4"	1,426	45,501	697
ga250c-5	333538	6"	2,602	82,561	976
ga500c-1	621360	44'31"	175,544	5,673,939	9,331
ga500c-2	621464	88'23"	328,682	10,324,601	12,499
ga500c-3	621428	101'56"	383,096	11,840,896	13,299
ga500c-4	621754	88'21"	312,472	9,600,336	11,919
gs250a-1	257964	51'12"	617,345	38,626,708	38,627
gs250a-2	257573	17'01"	199,171	12,187,024	18,473
gs250a-3	257626	95'32"	1,245,908	77,751,785	77,752
gs250a-4	257961	72'06"	964,562	59,278,305	59,279
gs250a-5	257896	101'45"	1,332,652	82,664,426	82,665
gs250b-1	276761	89'30"	1,659,142	82,929,645	82,930
gs250b-2	275675	11'49"	224,214	10,927,163	15,774
gs250b-3	275710	14'40"	276,632	13,602,650	17,729
gs250b-4	276114	5'14"	97,728	4,799,499	10,325
gs250b-5	275916	9'29"	173,536	8,815,952	14,381
gs250c-1	332935	4"	1,750	54,895	772
gs250c-2	334630	8"	3,476	110,912	1,152
gs250c-3	333000	6"	2,320	73,653	915
gs250c-4	333158	4"	1,568	49,979	739
gs250c-5	334635	13"	5,362	172,336	1,474
gs500c-1	620041	46'58"	185,670	5,850,888	9,368
gs500c-2	620434	46'40"	187,538	5,865,590	9,348
gs500c-4	620437	74'10"	300,514	9,343,075	11,822

Table 3.IV: Optimal values for the M* instance set.

instance	opt	time	#nodes	#lag.eval.	#moves
MO1	1305.95	1"	68	2,880	139
MO2	1432.36	1"	58	3,088	149
MO3	1516.77	1"	92	3,774	165
MO4	1442.24	1"	26	1,296	85
MO5	1408.77	1"	62	2,690	134
MP1	2686.48	1"	142	6,231	227
MP2	2904.86	1"	168	6,605	232
MP3	2623.71	1"	52	2,264	120
MP4	2938.75	1"	232	9,688	296
MP5	2932.33	1"	362	15,348	393
MQ1	4091.01	1"	108	4,436	182
MQ2	4028.33	1"	180	7,338	249
MQ3	4275.43	1"	106	4,256	177
MQ4	4235.15	1"	138	5,608	211
MQ5	4080.74	3"	608	25,728	537
MR1	2608.15	10"	890	38,963	692
MR2	2654.73	3"	280	9,187	275
MR3	2788.25	12"	1,582	48,423	714
MR4	2756.04	15"	1,744	61,719	855
MR5	2505.05	11"	1,142	44,873	732
MS1	5283.76	52"	1,090	37,352	635
MT1	10069.80	40'50"	20,094	663,007	3,080

CHAPITRE 4

COMPTE-RENDU – RESOLUTION SEARCH EN TANT QU'ALTERNATIVE AU BRANCH-AND-BOUND

Ce chapitre relie ensemble les précédents articles en expliquant comment nous avons tenté de résoudre les problèmes présentés dans les chapitres 2 et 3 avec une approche Resolution Search telle qu'introduite dans le chapitre 1.

Dans la section 4.1 de ce chapitre, nous introduisons un point commun à presque toutes les procédures `obstacle` développées pour nos diverses tentatives d'application de Resolution Search. Celui-ci consiste à réutiliser (recycler) des clauses nogood générées lors d'appels antérieurs de `obstacle`, quand c'est possible. Il s'agit d'un concept auquel nous avons déjà fait allusion vers la fin du chapitre 1. Ce recyclage de clauses est une généralisation d'une idée originellement présentée par Chvátal [11] dans le cas 0-1, mais qui n'avait pas non plus été présentée en détail.

Dans les sections 4.2 et 4.3, nous mettons en évidence les aspects les plus importants de l'application de Resolution Search dans le cas de deux problèmes académiques classiques : le problème d'affectation généralisé (GAP) et le problème de localisation simple (UFLP). Spécifiquement, pour chaque problème nous exposons comment nous pensions exploiter la structure de celui-ci pour générer des clauses nogood de manière efficace.

Enfin, la section 4.4 dresse le bilan de ces tentatives d'application.

4.1 Aperçu général de `obstacle` avec recyclage de clauses

Dans le chapitre 1, nous imposons les propriétés suivantes à la procédure `obstacle`, étant donné une famille path-like $\mathcal{F} = [C_1, \ldots, C_m]$ avec les prédicats marqués $\mathcal{M} = [\mu_1, \ldots, \mu_m]$ induisant une clause $U_{\mathcal{F}}$:

1. `obstacle`$(U_{\mathcal{F}})$ doit générer une clause S qui doit être nogood,
2. et telle que $X(U_{\mathcal{F}}) \cap X(S) \neq \emptyset$,

3. et enfin telle que S doit maintenir la structure path-like pour $U_{\mathcal{F}}$.

La dernière de ces trois conditions est beaucoup plus faible que les deux autres, comme le montre le résultat suivant. Rappelons d'abord qu'une clause S maintient la structure path-like pour $U_{\mathcal{F}}$ si et seulement si, pour tout prédicat $\gamma \in S$ marquable pour $U_{\mathcal{F}}$, nous avons $X(U_{\mathcal{F}}) \cap X(\bar{S}_\gamma)$ non vide, avec $\bar{S}_\gamma = ((S \setminus \{\gamma\}) \cup \{\bar{\gamma}\})$. Rappelons également qu'un prédicat γ est marquable pour une clause $U_{\mathcal{F}}$ si et seulement si $\gamma \notin U_{\mathcal{F}}$ et γ partitionne l'ensemble de solutions $\mathcal{X}$ en deux parties non vides.

Proposition 1. *Étant donné une clause $U_{\mathcal{F}}$ ainsi qu'une clause nogood N telle que $X(U_{\mathcal{F}}) \cap X(N) \neq \emptyset$, il est toujours possible d'identifier une clause nogood $S \subseteq (U_{\mathcal{F}} \cup N)$ telle que $X(U_{\mathcal{F}}) \cap X(S) \neq \emptyset$ et maintenant la structure path-like pour $U_{\mathcal{F}}$.*

Démonstration. Procédons par récurrence sur $N \setminus U_{\mathcal{F}}$. Considérons d'abord le cas où il n'existe aucun prédicat $\gamma \in N$ marquable pour $U_{\mathcal{F}}$ et tel que $X(U_{\mathcal{F}}) \cap X(\bar{N}_\gamma) = \emptyset$. Il suffit alors de poser $S \leftarrow N$.

Considérons maintenant le cas où il existe $\gamma \in N$ marquable pour $U_{\mathcal{F}}$ et vérifiant $X(U_{\mathcal{F}}) \cap X(\bar{N}_\gamma) = \emptyset$. Dans ce cas, γ ne peut pas être dans $U_{\mathcal{F}}$, alors posons $N' \leftarrow U_{\mathcal{F}} \cup (N \setminus \{\gamma\})$, et γ ne peut pas être dans N' non plus. Nous savons aussi que $X(U_{\mathcal{F}}) \cap X(N')$ est non vide, car $X(N' \cup \{\gamma\}) = X(U_{\mathcal{F}}) \cap X(N)$, qui est non vide aussi. Enfin, $N' \cup \{\gamma\}$ est une clause nogood car N est nogood, et comme $X(N' \cup \{\bar{\gamma}\}) = X(U_{\mathcal{F}}) \cup X(\bar{N}_\gamma) = \emptyset$, nous en déduisons que N' est nogood. Recommençons le même raisonnement en remplaçant N par N'. Nous obtenons une clause S adéquate en au plus $|N \setminus U_{\mathcal{F}}|$ itérations. □

En pratique, au lieu de procéder comme dans cette preuve, il est presque toujours possible d'exploiter directement la structure du problème à résoudre afin d'identifier une telle clause S étant données N et $U_{\mathcal{F}}$. Ceci est illustré par l'exemple suivant.

Exemple 1. Reprenons l'exemple du chapitre 1 et supposons que nous sommes au début d'une itération de Resolution Search avec $\bar{z} = -9$ et

$$\mathcal{F} = \begin{bmatrix} \{\underline{0 \leq l \leq 2}, 0 \leq h \leq 3\} \\ \{\underline{0 \leq l \leq 4}, 0 \leq h \leq 3\} \end{bmatrix}.$$

Cette famille path-like avec ces prédicats marqués induit $U_{\mathcal{F}} = \{3 \leq l \leq 5, 0 \leq h \leq 3, 5 \leq l \leq 5, 0 \leq h \leq 3\}$. La couverture de la clause nogood $N = \{4 \leq l \leq 5, 2 \leq h \leq 4\}$ a une intersection non vide avec celle de $U_{\mathcal{F}}$, car $X(U_{\mathcal{F}}) = \{(l, h) \in \mathbb{Z}^2 : 5 \leq l \leq 5, 0 \leq h \leq 3\}$, mais elle ne maintient pas la structure path-like pour $U_{\mathcal{F}}$. En effet le prédicat $(4 \leq l \leq 5)$ est marquable pour $U_{\mathcal{F}}$, mais l'intersection de $X(\{0 \leq l \leq 3, 2 \leq h \leq 4\})$ et de $X(U_{\mathcal{F}})$ est vide. De manière évidente, nous pouvons en tirer $S = \{5 \leq l \leq 5, 2 \leq h \leq 4\}$ et cette clause remplit les trois conditions sur `obstacle`. □

Par conséquent, il est possible de résoudre un problème avec Resolution Search étant donné seulement une procédure `gen_nogood`$(X(U_{\mathcal{F}}))$ qui renvoie une clause nogood N telle que $X(U_{\mathcal{F}}) \cap X(N) \neq \emptyset$ et qui met à jour la borne supérieure $\bar{z}$. Cette observation est sans intérêt en tant que telle, mais elle simplifie la spécification d'une procédure `obstacle` pratiquant le recyclage de clauses nogood, c'est-à-dire qui stocke toutes les clauses nogood générées par `gen_nogood` dans une collection $\mathcal{N}$ et qui les réutilise quand c'est possible.

Algorithme 1 - Procédure `obstacle`$(U_{\mathcal{F}})$:

---

1. Étant donné une collection de clauses nogood $\mathcal{N}$, chercher $N \in \mathcal{N}$ telle que $X(U_{\mathcal{F}}) \cap X(N) \neq \emptyset$.

2. Si N n'existe pas, générer $N \leftarrow$ `gen_nogood`$(X(U_{\mathcal{F}}))$ et ajouter N dans $\mathcal{N}$,

3. Générer et renvoyer une clause $S \subseteq (U_{\mathcal{F}} \cup N)$ telle que (1) S est nogood, (2) $X(U_{\mathcal{F}}) \cap X(S) \neq \emptyset$ et (3) S maintient la structure path-like pour $U_{\mathcal{F}}$.

Une telle approche n'a d'intérêt que si la recherche dans $\mathcal{N}$ dans l'étape 1 est suffisamment peu coûteuse computationellement par rapport à l'application de `gen_nogood` dans l'étape 2. Dans ce but, il s'avère nécessaire en pratique de périodiquement retirer des clauses de $\mathcal{N}$ afin de s'assurer que la collection ne grossit pas trop. Nous allons maintenant voir que la fouille dans l'étape 1 peut être aidée par un partitionnement judicieux de la collection $\mathcal{N}$ en autant de parties qu'il y a de clauses dans la famille path-like courante $\mathcal{F}$ plus une.

Remarquons en effet que

$$U_{\mathcal{F}_0} \subsetneq U_{\mathcal{F}_1} \subsetneq \cdots \subsetneq U_{\mathcal{F}_{m-1}} \subsetneq U_{\mathcal{F}_m} = U_{\mathcal{F}},$$

et donc que

$$X(U_{\mathcal{F}_m}) \subsetneq X(U_{\mathcal{F}_{m-1}}) \subsetneq \cdots \subsetneq X(U_{\mathcal{F}_1}) \subsetneq X(U_{\mathcal{F}_0}) = \mathcal{X},$$

avec pour tout k vérifiant $1 \leq k \leq m$, la clause $U_{\mathcal{F}_k}$ est celle induite par la sous-famille $\mathcal{F}_k = [C_1, \ldots, C_k]$ avec $\mathcal{M}_k = [\mu_1, \ldots, \mu_k]$. Nous définissons la partition $\mathcal{N}^0 \cup \mathcal{N}^1 \cup \ldots \cup \mathcal{N}^m$ de $\mathcal{N}$ de la manière suivante : pour toute clause nogood N dans $\mathcal{N}$, N est dans la partie $\mathcal{N}^j$ où j est le rang maximal tel que pour tout $0 \leq k \leq j$, $X(U_{\mathcal{F}_k}) \cap X(N) \neq \emptyset$.

L'intérêt d'une telle partition est double. D'une part, il suffit de ne fouiller que $\mathcal{N}^m$ dans l'étape 1 de `obstacle` telle que présentée dans l'algorithme 1. D'autre part, cette partition est relativement aisée à maintenir après la mise à jour de la famille path-like à la fin de l'itération de Resolution Search. En effet Resolution Search remplace la famille courante $\mathcal{F}$ soit par $\mathcal{F}' = [C_1, \ldots, C_m, S]$ soit par $\mathcal{F}' = [C_1, \ldots, C_k, R]$ avec $k < m$. Dans le premier cas, repartitionnons $\mathcal{N} = \mathcal{N}'^1 \cup \ldots \cup \mathcal{N}'^m \cup \mathcal{N}'^{m+1}$ avec $\mathcal{N}'^j = \mathcal{N}^j$ pour tout $j < m$, et en partitionnant $\mathcal{N}^m$ pour obtenir $\mathcal{N}'^m$ et $\mathcal{N}'^{m+1}$: pour toute clause $N \in \mathcal{N}^m$, mettons N dans $\mathcal{N}'^m$ si $X(U_{\mathcal{F}'}) \cap X(N) = \emptyset$ et dans $\mathcal{N}'^{m+1}$ sinon. Dans le second cas, procédons de même avec $(\mathcal{N}^{k+1} \cup \ldots \cup \mathcal{N}^m)$ à la place de $\mathcal{N}^m$.

Exemple 2. Comme dans l'exemple précédent, supposons que nous sommes au début d'une itération de Resolution Search avec $\bar{z} = -9$ et

$$\mathcal{F} = \begin{bmatrix} \{\underline{0 \leq l \leq 2}, 0 \leq h \leq 3\} \\ \{\underline{0 \leq l \leq 4}, 0 \leq h \leq 3\} \end{bmatrix} \begin{matrix} : C_1 \\ : C_2. \end{matrix}$$

Cette famille path-like avec ces prédicats marqués induit $U_{\mathcal{F}} = U_{\mathcal{F}_2} = \{3 \leq l \leq 5, 0 \leq h \leq 3, 5 \leq l \leq 5\}$ et $U_{\mathcal{F}_1} = \{3 \leq l \leq 5, 0 \leq h \leq 3\}$. Supposons maintenant que nous avons la collection de clauses suivante :

$$\mathcal{N} = \left\{ \begin{array}{l} \{0 \leq l \leq 2, 0 \leq h \leq 3\}, \\ \{0 \leq l \leq 4, 0 \leq h \leq 1\}, \\ \{4 \leq l \leq 5, 2 \leq h \leq 5\}, \\ \{0 \leq l \leq 3, 0 \leq h \leq 3\} \end{array} \right\} \begin{matrix} : N_1 \\ : N_2 \\ : N_3 \\ : N_4. \end{matrix}$$

Étant donné notre famille path-like $\mathcal{F}$ et notre choix de prédicats marqués, la collection $\mathcal{N}$ peut être partitionnée en 3 parties comme suit :

$$\mathcal{N} = \mathcal{N}^0 \cup \mathcal{N}^1 \cup \mathcal{N}^2 = \{N_1\} \cup \{N_2, N_4\} \cup \{N_3\}.$$

Comme $\mathcal{N}^2$ n'est pas vide, une application de `obstacle` sélectionne la clause $N_3 = \{4 \leq l \leq 5, 2 \leq h \leq 5\}$, évitant donc une application de `gen_nogood`. La clause N_3 est transformée en $\{5 \leq l \leq 5, 2 \leq h \leq 5\}$ comme expliqué dans l'exemple précédent avant d'être renvoyée à Resolution Search par `obstacle`$(U_{\mathcal{F}})$. □

Étant donné cette procédure `obstacle` et en mettant de côté les détails de la gestion de $\mathcal{N}$, il suffit de spécifier une procédure `gen_nogood` adéquate pour le problème d'optimisation combinatoire que nous souhaitons résoudre avec Resolution Search.

4.2 Cas du problème d'affectation généralisé

Comme expliqué dans les chapitres 1 et 2, le problème d'affectation généralisé consiste à affecter chaque tâche $j \in \{1, \ldots, n\}$ à un et un seul des m agents dans l'objectif de minimiser les coûts des affectations. Par contre, plusieurs tâches peuvent être affectées à un seul agent $i \in \{1, \ldots, m\}$, mais sa disponibilité globale est limitée par une contrainte qui prend la forme d'une inégalité knapsack : b_i unités de ressource sont disponibles, et l'affectation d'une tâche j en consomme a_{ij}. Rappelons que le problème peut être modélisé de la manière suivante :

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{m} \sum_{j=1}^{n} \mathbf{c}_{ij} \mathbf{x}_{ij} && && (GAP)\\
\text{s.à} \quad & \sum_{j=1}^{n} \mathbf{a}_{ij} \mathbf{x}_{ij} \leq \mathbf{b}_i, && 1 \leq i \leq m, && (1)\\
& \sum_{i=1}^{m} \mathbf{x}_{ij} = 1, && 1 \leq j \leq n, && (2)\\
& \mathbf{x} \in \{0,1\}^{mn}. && && (3)
\end{aligned}
$$

Rappelons également que l'ensemble des solutions pour ce problème $\mathcal{X}$ est égal à $\{0,1\}^{mn}$. Dans le chapitre 1, nous avons mentionné que des clauses pour ce problème pouvaient être constituées de prédicats de la forme $\left(\sum_{i \in I} \mathbf{x}_{ij} = 0\right)$, étant donné un sous-ensemble $I \subsetneq \{1, \ldots, m\}$ d'agents auxquels l'affectation d'une tâche $j \in \{1, \ldots, n\}$ est interdite. Une conséquence de ce choix de prédicats est que pour toute clause $U_{\mathcal{F}}$ ainsi constituée, l'ensemble $X(U_{\mathcal{F}})$ de solutions la vérifiant peut être caractérisé par la solution partielle $\mathbf{p} \in \{0, *\}^{mn}$ correspondante qui met en évidence les couples agent-tâche interdits par $U_{\mathcal{F}}$, et

$$
X(U_{\mathcal{F}}) = \left\{ \mathbf{x} \in \{0,1\}^{mn} : \forall (i,j) \in \{1, \ldots, m\} \times \{1, \ldots, n\}, \mathbf{p}_{ij} = 0 \implies \mathbf{x}_{ij} = 0 \right\}.
$$

Resolution Search nous garantit que $X(U_{\mathcal{F}})$ n'est pas vide, par conséquent pour tout $j \in \{1, \ldots, n\}$, il existe au moins un indice $i \in \{1, \ldots, m\}$ pour lequel $\mathbf{p}_{ij} = *$.

Définissons alors $GAP(\mathbf{p})$ comme étant le sous-problème suivant :

$$\begin{aligned}
\min \quad & \sum_{i=1}^{m}\sum_{j=1}^{n} \mathbf{c}_{ij}\mathbf{x}_{ij} && && (GAP(\mathbf{p}))\\
\text{s.à} \quad & \sum_{j=1}^{n} \mathbf{a}_{ij}\mathbf{x}_{ij} \leq \mathbf{b}_i, && 1 \leq i \leq m, && (1)\\
& \sum_{i=1}^{m} \mathbf{x}_{ij} = 1, && 1 \leq j \leq n, && (2)\\
& \mathbf{x} \in \{0,1\}^m, && && (3)\\
& \mathbf{x}_{ij} = 0 && \text{si } \mathbf{p}_{ij} = 0; \quad 1 \leq i \leq m, 1 \leq j \leq n. && (4)
\end{aligned}$$

Reprenons la relaxation lagrangienne introduite dans le chapitre 2 pour ce sous-problème, où les contraintes (2) sont dualisées avec un vecteur $\boldsymbol{\lambda} \in \mathbb{R}^n$:

$$z(\mathbf{p}, \boldsymbol{\lambda}) = \sum_{j=1}^{n} \boldsymbol{\lambda}_j + \sum_{i=1}^{m} \kappa^i(\mathbf{p}, \boldsymbol{\lambda})$$

avec, pour tout $i \in \{1, \ldots, m\}$,

$$\begin{aligned}
\kappa^i(\mathbf{p}, \boldsymbol{\lambda}) = \min \quad & \sum_{j=1}^{n} (\mathbf{c}_{ij} - \boldsymbol{\lambda}_j)\mathbf{x}_{ij}\\
\text{s.à} \quad & \sum_{j=1}^{n} \mathbf{a}_{ij}\mathbf{x}_{ij} \leq \mathbf{b}_i,\\
& \mathbf{x}_{ij} \in \{0,1\}, && 1 \leq j \leq n,\\
& \mathbf{x}_{ij} = 0 && \text{si } \mathbf{p}_{ij} = 0, \quad 1 \leq j \leq n.
\end{aligned}$$

Pour générer une clause nogood étant donné $X(U_{\mathcal{F}})$ (et par conséquent $\mathbf{p}$), commençons par optimiser le dual lagrangien

$$\bar{\boldsymbol{\lambda}} \approx \arg\max_{\boldsymbol{\lambda} \in \mathbb{R}^n} z(\mathbf{p}, \boldsymbol{\lambda}),$$

et considérons le cas où $z(\mathbf{p}, \bar{\boldsymbol{\lambda}})$ est plus grand ou égal qu'une borne supérieure $\bar{z}$. Dans ce cas, nous pourrions générer

$$N \leftarrow \left\{ \left(\sum_{\substack{i=1 \\ \mathbf{p}_{ij}=0}}^{m} \mathbf{x}_{ij} = 0 \right) : 1 \leq j \leq n \right\},$$

et N serait nogood en plus de vérifier $X(U_{\mathcal{F}}) \cap X(N) \neq \emptyset$, car $X(N) = X(U_{\mathcal{F}})$ non vide et qui est un sous-ensemble de solutions dont le coût dépasse une borne supérieure $\bar{z}$. Il est néanmoins possible de générer une clause N couvrant un plus grand sous-ensemble de $\mathcal{X}$ en exploitant les propriétés des coûts réduits générés de la manière présentée dans le chapitre 2. Désignons par $\mathbf{c}(\mathbf{p}, \bar{\boldsymbol{\lambda}})$ le vecteur de coûts réduits obtenus en résolvant la relaxation de $GAP(\mathbf{p})$ avec les multiplicateurs $\bar{\boldsymbol{\lambda}}$. Soit $\tilde{\mathbf{p}}$ le vecteur obtenu en prenant $\mathbf{p}$ et en mettant $\tilde{\mathbf{p}}_{ij}$ à $*$ pour tout couple agent-tâche (i, j) pour lequel $\mathbf{p}_{ij} = 0$ et pour lequel le coût réduit correspondant est négatif. Générons N plutôt de la manière suivante, avec $\tilde{\mathbf{p}}$ à la place de $\mathbf{p}$:

$$N \leftarrow \left\{ \left(\sum_{\substack{i=1 \\ \tilde{\mathbf{p}}_{ij}=0}}^{m} \mathbf{x}_{ij} = 0 \right) : 1 \leq j \leq n \right\}.$$

Cette clause est nogood et vérifie $X(U_{\mathcal{F}}) \cap X(N) \neq \emptyset$. En effet, nous avons $z(\tilde{\mathbf{p}}, \bar{\boldsymbol{\lambda}}) = z(\mathbf{p}, \bar{\boldsymbol{\lambda}})$ et par conséquent toute solution vérifiant N coûte plus cher que $\bar{z}$, et de plus $X(N) \subseteq X(U_{\mathcal{F}}) \neq \emptyset$.

Dans le cas où $z(\mathbf{p}, \bar{\boldsymbol{\lambda}}) < \bar{z}$ et qu'il existe un couple agent-tâche (i, j) pour lequel $\mathbf{p}_{ij} = *$ et $z(\mathbf{p}, \bar{\boldsymbol{\lambda}}) + \mathbf{c}(\mathbf{p}, \bar{\boldsymbol{\lambda}})_{ij} \geq \bar{z}$, nous pouvons également construire une clause nogood directement d'après les informations duales que nous avons obtenues. Désignons par $\mathbf{x}(\mathbf{p}, \bar{\boldsymbol{\lambda}})$ la solution optimale que nous avons calculée pour la relaxation lagrangienne de $GAP(\mathbf{p})$ avec le vecteur de multiplicateurs $\bar{\boldsymbol{\lambda}}$. Soit $\tilde{\mathbf{p}}$ le vecteur obtenu en prenant $\mathbf{p}$ et en fixant soit $\tilde{\mathbf{p}}_{ij}$ à 0 si $\mathbf{x}(\mathbf{p}, \bar{\boldsymbol{\lambda}})_{ij} = 1$, soit $\tilde{\mathbf{p}}_{kj}$ à 0 pour tout $k \neq i$ si $\mathbf{x}(\mathbf{p}, \bar{\boldsymbol{\lambda}}) = 0$. Nous pouvons alors générer une clause nogood N comme

dans le paragraphe précédent. Nous pouvons aussi procéder similairement avec les autres règles de fixation de variable introduites dans le chapitre 2.

Dans le cas restant où $z(\mathbf{p}, \bar{\boldsymbol{\lambda}}) < \bar{z}$ et où nous ne pouvons appliquer aucune règle de fixation, nous pouvons générer une clause nogood à partir d'une solution $\tilde{\mathbf{x}} \in X(U_{\mathcal{F}})$ obtenue en réparant la solution de la relaxation lagrangienne $\mathbf{x}(\mathbf{p}, \bar{\boldsymbol{\lambda}})$ par recherche locale (voir Haddadi et Ouzia [24]). Supposons que solution $\tilde{\mathbf{x}}$ ainsi obtenue satisfait au moins les contraintes d'affectation (2). Deux sous-cas se présentent : soit $\tilde{\mathbf{x}}$ satisfait toutes les contraintes de ressource (1), soit pas.

Dans le sous-cas où la solution $\tilde{\mathbf{x}}$ satisfait toutes les contraintes de ressource, celle-ci est réalisable et nous devons mettre à jour $\bar{z} \leftarrow \min\left\{ \bar{z},\ \sum_{i=1}^{m} \sum_{j=1}^{n} \mathbf{c}_{ij} \tilde{\mathbf{x}}_{ij} \right\}$. Pour toute tâche $j \in \{1, \ldots, n\}$, désignons par $\boldsymbol{\alpha}_j$ l'agent à laquelle celle-ci a été affectée dans la solution $\tilde{\mathbf{x}}$, posons $\boldsymbol{\lambda}_j \leftarrow \mathbf{c}_{\boldsymbol{\alpha}_j j}$, et pour tout $i \in \{1, \ldots, m\}$, posons $\tilde{\mathbf{p}}_{ij} \leftarrow 0$ si $\mathbf{c}_{ij} \leq \boldsymbol{\lambda}_j$ et si $i \neq \boldsymbol{\alpha}_j$, et $\tilde{\mathbf{p}}_{ij} \leftarrow *$ sinon. Remarquons que $\tilde{\mathbf{x}}$ est une solution optimale de la relaxation lagrangienne de $GAP(\tilde{\mathbf{p}})$ avec les multiplicateurs $\boldsymbol{\lambda}$ et donc que $z(\tilde{\mathbf{p}}, \boldsymbol{\lambda}) \geq \bar{z}$. Par conséquent, la clause N générée à partir de cette solution partielle $\tilde{\mathbf{p}}$ est nogood et vérifie $X(U_{\mathcal{F}}) \cap X(N) \neq \emptyset$.

Dans l'autre sous-cas, il existe un agent $i \in \{1, \ldots, m\}$ pour lequel la solution $\tilde{\mathbf{x}}$ ne satisfait pas la contrainte de ressource. En d'autres termes, nous avons $\sum_{j=1}^{n} \mathbf{a}_{ij} \tilde{\mathbf{x}}_{ij} > \mathbf{b}_i$. Prenons $J \subseteq \{1, \ldots, n\}$ le sous-ensemble de tâches pour lesquelles le terme $\tilde{\mathbf{x}}_{ij}$ est à 1 dans cette inégalité, et celle-ci devient $\sum_{j \in J} \mathbf{a}_{ij} > \mathbf{b}_i$. Calculons un sous-ensemble $J' \subseteq J$ qui est irréductible, c'est-à-dire tel que $\sum_{j \in J'} \mathbf{a}_{ij} > \mathbf{b}_i$ et tel que pour tout $J'' \subsetneq J'$, $\sum_{j \in J''} \mathbf{a}_{ij} \leq \mathbf{b}_i$. Pour tout couple agent-tâche (i, j), posons $\tilde{\mathbf{p}}_{ij} \leftarrow 0$ si $j \in J'$ et si $\tilde{\mathbf{x}}_{ij} = 0$, et $\tilde{\mathbf{p}}_{ij} \leftarrow *$ sinon. Ici encore, la clause N générée à partir de cette solution partielle $\tilde{\mathbf{p}}$ est nogood et vérifie $X(U_{\mathcal{F}}) \cap X(N) \neq \emptyset$.

4.3 Cas du problème de localisation simple

Le problème de localisation simple, traité dans le chapitre 3, consiste à décider pour n différents endroits de l'ouverture d'un dépot ou non, afin d'approvisionner m clients à moindre coût. L'approvisionnement d'un client j à partir d'un dépôt

situé à l'endroit i entraîne un coût $\mathbf{s}_{ij}$, et l'ouverture d'un dépôt en i entraine un coût $\mathbf{c}_i$: il s'agit donc de trouver le meilleur compromis entre coûts d'ouverture et coûts d'approvisionnement. Les mécanismes de génération de clause nogood décrits dans cette section reprennent le même schéma que dans la section précédente, notamment en ce qui concerne l'exploitation des coûts réduits. En revanche, contrairement à la section précédente, le mécanisme de génération de clauses pour le problème de localisation simple que nous allons présenter génère des clauses 0-1 similaires à celles dans [11].

Le chapitre 3 présente une méthode pour résoudre ce problème exactement, où une métaheuristique (processus primal) et un branch-and-bound (processus dual) coopèrent en s'échangeant de l'information. La métaheuristique a pour rôle d'améliorer la meilleure solution connue, et quand elle en trouve une, elle communique sa valeur au branch-and-bound qui améliore sa borne supérieure. Le branch-and-bound quant à lui a pour rôle de prouver l'optimalité de la meilleure solution connue. Notre approche visait originellement à utiliser une méthode Resolution Search au lieu d'un branch-and-bound. Voyons maintenant diverses manières de générer des clauses nogood pour ce problème, en réutilisant la notation et les concepts introduits dans le chapitre 3.

Rappelons que le problème de localisation simple peut être modélisé de la manière suivante :

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{n} \mathbf{c}_i \mathbf{x}_i + \sum_{j=1}^{m} \sum_{i=1}^{n} \mathbf{s}_{ij} \mathbf{y}_{ij} & & \\
\text{s.à} \quad & \sum_{i=1}^{n} \mathbf{y}_{ij} = 1, & 1 \leq j \leq m, & \quad (1) \\
& 0 \leq \mathbf{y}_{ij} \leq \mathbf{x}_i, & 1 \leq i \leq n, 1 \leq j \leq m, & \quad (2) \\
& \mathbf{x} \in \{0,1\}^n. & & \quad (3)
\end{aligned}
$$

Rappelons également que $SP(\mathbf{p}, \underline{n}, \overline{n})$ avec $\mathbf{p} \in \{0,1,*\}^n$ et $1 \leq \underline{n} \leq \overline{n} \leq n$ dénote

le sous-problème suivant :

$$\begin{aligned}
\min \quad & \sum_{i=1}^{n} \mathbf{c}_i \mathbf{x}_i + \sum_{j=1}^{m} \sum_{i=1}^{n} \mathbf{s}_{ij} \mathbf{y}_{ij} && && (SP(\mathbf{p}, \underline{n}, \overline{n})) \\
\text{s.à} \quad & \sum_{i=1}^{n} \mathbf{y}_{ij} = 1, && 1 \leq j \leq m, && (1) \\
& 0 \leq \mathbf{y}_{ij} \leq \mathbf{x}_i, && 1 \leq i \leq n, 1 \leq j \leq m, && (2) \\
& \mathbf{x} \in \{0,1\}^n, && && (3) \\
& \mathbf{x}_i = \mathbf{p}_i, && 1 \leq i \leq n, \ \mathbf{p}_i \in \{0,1\}, && (4) \\
& \underline{n} \leq \sum_{i=1}^{n} \mathbf{x}_i \leq \overline{n}. && && (5)
\end{aligned}$$

Rappelons finalement qu'une borne inférieure pour la solution optimale à ce sous-problème peut être calculée à l'aide de sa relaxation lagrangienne $LR(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu})$, obtenue en dualisant les contraintes (1) à l'aide d'un vecteur de multiplicateurs $\boldsymbol{\mu}$. Dénotons $\phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu})$ sa valeur objectif :

$$\begin{aligned}
\phi(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu}) = \sum_{j=1}^{m} \boldsymbol{\mu}_j + \ \min \quad & \sum_{i=1}^{n} \bar{\mathbf{c}}_i^{\boldsymbol{\mu}} \, \mathbf{x}_i && && (LR(\mathbf{p}, \underline{n}, \overline{n}, \boldsymbol{\mu})) \\
\text{s.à} \quad & \underline{n} \leq \sum_{i=1}^{n} \mathbf{x}_i \leq \overline{n}, && && (5) \\
& \mathbf{x}_i = \mathbf{p}_i, && 1 \leq i \leq n, \ \mathbf{p}_i \in \{0,1\}, && (4) \\
& \mathbf{x} \in \{0,1\}^n, && && (3)
\end{aligned}$$

avec

$$\bar{\mathbf{c}}_i^{\boldsymbol{\mu}} \ = \ \mathbf{c}_i + \sum_{j=1}^{m} \min \left\{ 0, \mathbf{s}_{ij} - \boldsymbol{\mu}_j \right\}.$$

Pour ce problème, nous avons $\mathcal{X} = \{0,1\}^n$, c'est-à-dire l'espace des variables $\mathbf{x}$. Les variables $\mathbf{y}$ sont en quelque sorte des variables auxiliaires. Supposons que

$X(U_{\mathcal{F}})$ peut être caractérisé par un vecteur associé $\mathbf{p} \in \{0, 1, *\}^n$ tel que

$$X(U_{\mathcal{F}}) = \{\mathbf{x} \in \{0,1\}^n : \mathbf{x}_i = \mathbf{p}_i, \forall i \in \{1, \ldots, n\}, \mathbf{p}_i \in \{0,1\}\}.$$

Optimisons le dual lagrangien du sous-problème correspondant et notons $\bar{\boldsymbol{\mu}}$ le meilleur vecteur de multiplicateurs trouvé :

$$\bar{\boldsymbol{\mu}} \approx \arg\max_{\boldsymbol{\mu} \in \mathbb{R}^m} \phi(\mathbf{p}, 1, n, \boldsymbol{\mu}).$$

Considérons d'abord le cas où ce maximum est supérieur ou égal à une borne supérieure $\bar{z}$. Dans ce cas, nous pouvons facilement générer la clause nogood suivante à partir de $\mathbf{p}$:

$$\{(\mathbf{x}_i = \mathbf{p}_i) : \forall i \in \{1, \ldots, n\}, \mathbf{p}_i \in \{0,1\}\}.$$

En effet, toute solution $\mathbf{x}$ vérifiant cette clause est nécessairement dans $X(U_{\mathcal{F}})$ et donc coûte au moins $\bar{z}$. Il est néanmoins possible de faire mieux, en exploitant les propriétés des coûts réduits $\bar{\mathbf{c}}^{\bar{\boldsymbol{\mu}}}$. En effet, il n'est pas nécessaire de fixer une variable $\mathbf{x}_i$ à 0 dans la clause si son coût réduit $\bar{\mathbf{c}}_i^{\bar{\boldsymbol{\mu}}}$ n'est pas négatif, et de façon similaire de fixer $\mathbf{x}_i$ à 1 si $\bar{\mathbf{c}}_i^{\bar{\boldsymbol{\mu}}} \leq 0$. Soit $\tilde{\mathbf{p}}$ le vecteur obtenu en prenant $\mathbf{p}$ et en défixant toutes ses composantes qui vérifient ces propriétés. Nous avons alors $\phi(\tilde{\mathbf{p}}, 1, n, \bar{\boldsymbol{\mu}}) = \phi(\mathbf{p}, 1, n, \bar{\boldsymbol{\mu}})$. Il est possible d'aller plus loin et de défixer d'autres composantes de $\tilde{\mathbf{p}}$ tant que $\phi(\tilde{\mathbf{p}}, 1, n, \bar{\boldsymbol{\mu}}) \geq \bar{z}$, mais l'idée générale reste la même. Ayant obtenu $\tilde{\mathbf{p}}$ nous générons la clause nogood correspondante.

$$N \leftarrow \{(\mathbf{x}_i = \tilde{\mathbf{p}}_i) : \forall i \in \{1, \ldots, n\}, \tilde{\mathbf{p}}_i \in \{0,1\}\}.$$

Considérons maintenant le cas où $\phi(\mathbf{p}, 1, n, \bar{\boldsymbol{\mu}}) < \bar{z}$. Pour tout $k \in \{1, \ldots, n\}$ pour lequel $\mathbf{p}_k = *$, notons $\mathbf{p}^{+k}$ le vecteur obtenu en prenant $\mathbf{p}$ et en fixant sa k-ième composante à 1, et notons $\mathbf{p}^{-k}$ le même vecteur mais avec 0 au lieu de 1. Nous avons vu dans la sous-section 3.3.3 du chapitre 3 comment calculer facilement $\phi(\mathbf{p}^{\pm k}, 1, n, \bar{\boldsymbol{\mu}})$. Il s'ensuit que si, pour un k donné, nous avons $\phi(\mathbf{p}^{+k}, 1, n, \bar{\boldsymbol{\mu}}) \geq \bar{z}$,

alors nous pouvons facilement générer la clause nogood suivante à partir de $\mathbf{p}^{+k}$ comme dans le paragraphe précédent, et vice-versa avec $\mathbf{p}^{-k}$. En revanche, si nous n'avons pu générer aucune clause nogood de cette manière, nous pouvons faire appel à la métaheuristique du processus primal.

Rappelons que le processus primal est une métaheuristique du type tabou, qui modifie une solution $\check{\mathbf{x}}$ en changeant le statut d'un endroit à la fois. Pour y arriver nous utilisons le vecteur $\boldsymbol{\delta}$ continuellement mis à jour tel que chaque composante $\boldsymbol{\delta}_i$ représente la variation du coût $f(\check{\mathbf{x}})$ suite à l'inversion du $i$ème bit de $\check{\mathbf{x}}$.

Supposons qu'après un certain nombre d'inversions, la solution $\check{\mathbf{x}}$ est telle que $\check{\mathbf{x}}_i = \mathbf{p}_i$ pour tout indice $i \in \{1, \ldots, n\}$ pour lequel $\mathbf{p}_i \in \{0, 1\}$. Nous avons alors $\check{\mathbf{x}} \in X(U_{\mathcal{F}})$, et nous pouvons générer une clause nogood N telle que $X(U_{\mathcal{F}}) \cap X(N) \neq \emptyset$ en initialisant $N \leftarrow \emptyset$, et puis pour tout $i \in \{1, \ldots, n\}$,

- si $\check{\mathbf{x}}_i = 0$ et $\boldsymbol{\delta}_i < 0$, alors $N \leftarrow N \cup \{\mathbf{x}_i = 0\}$,
- si $\check{\mathbf{x}}_i = 1$, alors $N \leftarrow N \cup \{\mathbf{x}_i = 1\}$.

Si nous mettons à jour la borne supérieure $\bar{z} \leftarrow \min\{\bar{z}, f(\check{\mathbf{x}})\}$, alors d'après la propriété suivante la clause N est nogood.

Proposition 2. *Pour tout* $\mathbf{x} \in X(N)$, $f(\check{\mathbf{x}}) \leq f(\mathbf{x})$.

Démonstration. Par construction de la clause N, la solution $\mathbf{x}$ peut être obtenue en prenant $\check{\mathbf{x}}$ et en inversant succesivement les composantes à 0 dans $\check{\mathbf{x}}$ mais à 1 dans $\mathbf{x}$. Dénotons par I l'ensemble des indices correspondants, et prenons $k \in I$ quelconque.

Soit $\check{\mathbf{x}}^k$ la solution obtenue en prenant $\check{\mathbf{x}}$ et en modifiant sa $k$ème composante avec $k \in \{1, \ldots, n\}$, et soit $\boldsymbol{\delta}^k$ son vecteur de différence de coûts correspondant. Nous savons que $\boldsymbol{\delta}^k_i \geq 0$ pour tout $i \in I$, et puisque $k \in I$, nous en déduisons que $f(\check{\mathbf{x}}^k) \geq f(\check{\mathbf{x}})$.

De plus, pour tout indice $i \in I \setminus \{k\}$, nous avons

$$\boldsymbol{\delta}_i = \mathbf{c}_i + \sum_{j=1}^{m} \min\left\{0, \min\{\mathbf{s}_{i'j} : 1 \leq i' \leq n, \check{\mathbf{x}}_{i'} = 1\} - \mathbf{s}_{ij}\right\}$$

et

$$\begin{aligned}\boldsymbol{\delta}_i^k &= \mathbf{c}_i + \sum_{j=1}^{m} \min\left\{0, \min\{\mathbf{s}_{i'j} : 1 \leq i' \leq n, \check{\mathbf{x}}_{i'}^k = 1\} - \mathbf{s}_{ij}\right\} \\ &= \mathbf{c}_i + \sum_{j=1}^{m} \min\left\{0, \min\{\mathbf{s}_{i'j} : 1 \leq i' \leq n, \check{\mathbf{x}}_{i'} = 1\} \cup \{\mathbf{s}_{kj}\} - \mathbf{s}_{ij}\right\}.\end{aligned}$$

Par conséquent, $\boldsymbol{\delta}_i^k \geq \boldsymbol{\delta}_i$. Retirons k de I, ainsi par induction sur k nous obtenons $f(\mathbf{x}) \geq f(\check{\mathbf{x}})$. □

4.4 Bilan

Dans chacune des deux précédentes sections, nous avons présenté des méchanismes de génération de clauses nogood pour une procédure `gen_nogood` avec laquelle Resolution Search (avec `obstacle` décrite dans la section 4.1) pourrait résoudre le problème correspondant de manière efficace. En plus du problème d'affectation généralisée et du problème de localisation simple, nous avons essayé de développer des procédures `gen_nogood` pour plusieurs autres problèmes : le problème d'ordonnancement d'ateliers (*job-shop scheduling problem*), le problème de classification en deux groupes (*two-group classification problem*), le problème de conception de réseau multi-commodités avec contraintes de capacité (*capacitated multicommodity network design problem*), ainsi que la programmation linéaire en nombres entiers. Les tests préliminaires que nous avons effectués lors de ces travaux n'ont pas été prometteurs.

Les particularités de chacun de ces échecs ne sont guère intéressantes en tant que telles, en revanche nous avons rencontré plusieurs difficultés communes durant toutes ces tentatives. Nous pouvons les poser sous la forme des questions ouvertes suivantes qui portent sur Resolution Search :

- À la fin de chaque itération, lors de l'ajout d'une clause nogood à la famille path-like, comment choisir le prédicat de cette clause à marquer ?
- Lors de chaque appel de `obstacle`, nous commençons par fouiller la partie

$\mathcal{N}^m$ de la collection $\mathcal{N}$ de clauses nogood. Si $\mathcal{N}^m$ contient plusieurs clauses, laquelle choisir ?

- En pratique, la collection $\mathcal{N}$ ne peut pas croître indéfiniment. Dans ce cas, quelles clauses en retirer, et quand ?

Nous n'avons pas trouvé de réponses évidentes à ces questions, et pourtant ces choix stratégiques sont cruciaux. Nous pouvons dresser un parallèle avec le Branch-and-bound, où une question similaire est : comment choisir la variable de branchement ? Comme de nombreuses recherches l'ont montré (c.f. la revue de Atamtürk et Savelsbergh [1]) la stratégie utilisée pour choisir la variable de branchement a un fort impact sur les performances des logiciels solveurs de problèmes de programmation linéaire en nombres entiers. De plus, les stratégies performantes sont souvent contre-intuitives, et les performances sont difficiles sinon impossibles à estimer autrement qu'au cas-par-cas. Il n'est donc pas farfelu qu'il en aille de même pour Resolution Search. Contrairement au Branch-and-bound qui a été très employé et bien étudié depuis plus de 40 ans, nous partons ici de zéro dans l'élaboration des trois stratégies listées plus haut.

Une autre difficulté commune à toutes nos tentatives d'application de Resolution Search concerne la réplication d'effort computationnel. Rappelons que dans une méthode de type Branch-and-bound nous pouvons évaluer un noeud en partant des résultats de l'évaluation de son noeud parent. Par exemple, en programmation linéaire en nombres entiers, la relaxation linéaire d'un noeud est optimisée par l'algorithme du simplexe dual, qui est initialisé avec la base duale optimale obtenue pour le parent. Le caractère *bottom-up* de Resolution Search ne nous permet hélas pas d'en faire autant.

Enfin, Resolution Search (avec le recyclage de clauses nogood) n'a pas non plus l'avantage de la simplicité, contrairement au Branch-and-bound qui met en oeuvre le principe de diviser-pour-régner de manière assez naturelle. D'un point de vue purement pratique, cette complexité ouvre la porte à beaucoup d'erreurs d'implémentation.

Le bilan global de nos recherches est donc que malgré notre contribution théorique, nous n'avons pas su appliquer Resolution Search de manière pratique pour résoudre des problèmes d'optimisation discrète.

CONCLUSION

Les deux objectifs principaux de cette thèse étaient de mieux comprendre Resolution Search et de l'appliquer de manière pratique à résoudre des problèmes d'optimisation discrète.

Sur le plan théorique, notre principale contribution est la généralisation de Resolution Search, qui était initialement spécifiée pour des problèmes à variables 0-1. D'une part, nous avons étendu l'approche aux problèmes d'optimisation discrète en général, et avons de plus relaché la spécification des procédures `obstacle` qui explorent l'espace de solutions. D'autre part et de manière subséquente, nous avons étendu la technique de recyclage de clauses et avons présenté un mécanisme général de stockage et de réutilisation de clauses nogood. Ceci nous a permis de séparer l'aspect génération de nouvelles clauses nogood de l'aspect gestion de clauses nogood générées précédemment.

Sur le plan pratique, nos tentatives d'application ont abouti à des méthodes efficaces qui, en revanche, ne font pas appel à Resolution Search. En effet, une application de Resolution Search nécéssite invariablement l'élaboration de nombreuses stratégies de recherche qui ne sont pas évidentes, dans le sens où il est difficile d'évaluer la pertinence d'un choix sur un autre, dans le cadre de la résolution d'un problème donné. Cela complique le développement d'une telle méthode au point où nous finissons parfois par trouver des alternatives plus simples et plus efficaces, comme cela a été le cas en particulier pour le problème d'affectation généralisé et le problème de localisation simple.

La question de comment appliquer Resolution Search de manière efficace reste donc entièrement ouverte, et plusieurs pistes de recherche se dessinent devant nous. Nous pourrions par exemple chercher à appliquer Resolution Search dans un contexte différent de l'optimisation combinatoire, comme par exemple pour résoudre des problèmes de satisfaisabilité. Ceux-ci ont en effet l'avantage d'avoir été très étudiés, et plusieurs méthodes très efficaces ont été proposées qui se basent déjà sur l'apprentissage de clauses nogood. Une autre piste consisterait à aban-

donner l'exactitude de la recherche et à chercher à faire fonctionner Resolution Search comme une métaheuristique tabou, où la famille path-like ferait office de liste tabou. Enfin, peut-être que nous pourrions développer d'autres approches de résolution exacte de nature *bottom-up*, c'est à dire qui s'appuyent sur la génération de clauses nogood plutôt que sur le partitionnement récursif de l'espace de solutions.

La contribution de cette thèse pour le problème d'affectation généralisé consiste en une méthode exacte plutôt simple qui améliore sensiblement l'état de l'art. Le problème d'optimisation est transformé en une suite de problèmes de décision de la forme « existe-t'il une solution réalisable de coût inférieur un égal à une valeur $\bar{z}$? », où $\bar{z}$ est successivement incrémentée jusqu'à l'obtention d'une solution réalisable qui est alors optimale. Pour que la méthode soit efficace, nous introduisons un mécanisme de fixation de variable se basant sur des coûts réduits lagrangiens obtenus par programmation dynamique. Notre méthode donne d'excellents résultats pour ce problème, aussi peut-être serait-il intéressant de la transposer à d'autres problèmes ayant des caractéristiques similaires à celui-ci.

La contribution de cette thèse pour le problème de localisation simple consiste en une méthode exacte coopérative qui s'avère être efficace sur des instances de nature diverse et variée. Notre méthode allie un processus appliquant une recherche locale à un autre processus appliquant un Branch-and-bound, et les fait s'échanger de l'information. Une contribution particulière pour le Branch-and-bound consiste en la séparation de l'espace de solutions selon le nombre de dépôts ouverts, en plus de l'ouverture ou non d'un dépôt sur un site potentiel, ce qui permet de mieux élaguer l'arbre de recherche. Cette méthode est trivialement parallélisable mais ne comporte que deux processus, aussi peut-être pourrait-on s'intéresser à pousser la parallélisation plus loin, avec par exemple plusieurs processus de recherche métaheuristique. Nous pourrions également avoir plusieurs processus de recherche exacte, par exemple en décomposant le problème en n sous-problèmes où le nombre de dépôts à ouvrir est fixé, et lancer un processus par sous-problème. En effet, si l'avenir sera bien constitué de processeurs massivement multi-coeurs comme les

prédictions actuelles le laissent entendre, une telle approche pourrait devenir plus intéressante qu'elle ne l'est actuellement.

BIBLIOGRAPHIE

[1] A. Atamtürk et M.W.P. Savelsbergh. Integer-programming software systems. *Annals of Operations Research*, 140(1):67–124, 2005.

[2] P. Avella, M. Boccia et I. Vasilyev. A computational study of exact knapsack separation for the generalized assignment problem. *Computational Optimization and Applications*, 45(3):543–555, 2010.

[3] F. Barahona et R. Anbil. The volume algorithm : producing primal solutions with a subgradient method. *Mathematical Programming*, 87(3):385–399, 2000.

[4] E.M.L. Beale et J.A. Tomlin. Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables. *Mathematical Methods of Optimization*, 69:447–454, 1970.

[5] J.E. Beasley. Generalised assignment problem test data sets. `http://people.brunel.ac.uk/~mastjjb/jeb/orlib/gapinfo.html`, 2011.

[6] C. Beltran, C. Tadonki et J.P. Vial. Solving the p-median problem with a semi-lagrangian relaxation. *Computational Optimization and Applications*, 35 (2):239–260, 2006.

[7] C. Beltran-Royo, J.P. Vial et A. Alonso-Ayuso. Semi-lagrangian relaxation applied to the uncapacitated facility location problem. *Computational Optimization and Applications*, pages 1–23, 2010.

[8] O. Bilde et J. Krarup. Sharp lower bounds and efficient algorithms for the simple plant location problem. *Annals of Discrete Mathematics*, 1:79–97, 1977.

[9] B.H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970. ISSN 0001-0782.

[10] S. Boussier, M. Vasquez, Y. Vimont, S. Hanafi et P. Michelon. A multi-level search strategy for the 0-1 multidimensional knapsack problem. *Discrete Applied Mathematics*, 158(2):97–109, 2010.

[11] V. Chvátal. Resolution search. *Discrete Applied Mathematics*, 73(1):81–99, 1997.

[12] T. Cura. A parallel local search approach to solving the uncapacitated warehouse location problem. *Computers and Industrial Engineering*, 59(4):1000–1009, 2010. ISSN 0360-8352.

[13] S. Demassey, C. Artigues et P. Michelon. An application of resolution search to the rcpsp. Dans *17th European Conference on Combinatorial Optimization ECCO*, 2004.

[14] J.A. Diaz et E. Fernandez. A tabu search heuristic for the generalized assignment problem. *European Journal of Operational Research*, 132(1):22–38, 2001.

[15] D. Erlenkotter. A dual-based procedure for uncapacitated facility location. *Operations Research*, 26(6):992–1009, 1978. ISSN 0030-364X.

[16] A.D. Flaxman, A.M. Frieze et J.C. Vera. On the average case performance of some greedy approximation algorithms for the uncapacitated facility location problem. *Combinatorics, Probability and Computing*, 16(05):713–732, 2007. ISSN 0963-5483.

[17] A. Frangioni. Solving semidefinite quadratic problems within nonsmooth optimization algorithms. *Computers and Operations Research*, 23(11):1099–1118, 1996.

[18] R.D. Galvão et L.A. Raggi. A method for solving to optimality uncapacitated location problems. *Annals of Operations Research*, 18(1):225–244, 1989.

[19] D. Ghosh. Neighborhood search heuristics for the uncapacitated facility location problem. *European Journal of Operational Research*, 150(1):150–162, 2003. ISSN 0377-2217.

[20] F. Glover. A template for scatter search and path relinking. Dans *Artificial evolution*, pages 1–51. Springer, 1998.

[21] B. Goldengorin, D. Ghosh et G. Sierksma. Branch and peg algorithms for the simple plant location problem. Dans *Computers and Operations Research*. Citeseer, 2003.

[22] B. Goldengorin, G.A. Tijssen, D. Ghosh et G. Sierksma. Solving the simple plant location problem using a data correcting approach. *Journal of Global Optimization*, 25(4):377–406, 2003.

[23] A.R. Guner et M. Sevkli. A discrete particle swarm optimization algorithm for uncapacitated facility location problem. *Journal of Artificial Evolution and Applications*, 2008:1–9, 2008. ISSN 1687-6229.

[24] S. Haddadi et H. Ouzia. Effective algorithm and heuristic for the generalized assignment problem. *European Journal of Operational Research*, 153(1):184–190, 2004.

[25] S. Hanafi et F. Glover. Resolution search and dynamic branch-and-bound. *Journal of combinatorial optimization*, 6(4):401–423, 2002.

[26] P. Hansen, J. Brimberg, D. Urošević et N. Mladenović. Primal-dual variable neighborhood search for the simple plant-location problem. *INFORMS Journal on Computing*, 19(4):552, 2007.

[27] WC Healy. Mutiple choice programming. *Operations Research*, page 12, 1964.

[28] J.B. Hiriart-Urruty et C. Lemaréchal. *Convex analysis and minimization algorithms : Fundamentals*, volume 305. Springer-Verlag, 1996.

[29] J. Homberger et H. Gehring. A Two-Level Parallel Genetic Algorithm for the Uncapacitated Warehouse Location Problem. Dans *Hawaii International Conference on System Sciences, Proceedings of the 41st Annual*, page 67. IEEE, 2008.

[30] B. Julstrom. A Permutation Coding with Heuristics for the Uncapacitated Facility Location Problem. *Recent Advances in Evolutionary Computation for Combinatorial Optimization*, pages 295–307, 2008.

[31] N. Karabakal, J.C. Bean et J.R. Lohmann. A steepest descent multiplier adjustment method for the generalized assignment problem. Rapport technique, University of Michigan, 1992.

[32] M. Körkel. On the exact solution of large-scale simple plant location problems. *European Journal of Operational Research*, 39(2):157–173, 1989. ISSN 0377-2217.

[33] J. Kratica, D. Tošic, V. Filipović et I. Ljubić. Solving the simple plant location problem by genetic algorithm. *Operations Research*, 35(1):127–142, 2001. ISSN 0399-0559.

[34] A.N. Letchford et S.J. Miller. Fast bounding procedures for large instances of the simple plant location problem. *Computers and Operations Research*, 2011.

[35] L. Michel et P. Van Hentenryck. A simple tabu search for warehouse location. *European Journal of Operational Research*, 157(3):576–591, 2004. ISSN 0377-2217.

[36] N. Mladenović, J. Brimberg et P. Hansen. A note on duality gap in the simple plant location problem. *European Journal of Operational Research*, 174(1): 11–22, 2006. ISSN 0377-2217.

[37] İ. Muter, S.I. Birbil et G. Sahin. Combination of metaheuristic and exact algorithms for solving set covering-type optimization problems. *INFORMS Journal on Computing*, 22(4):603–619, 2010.

[38] R.M. Nauss. Solving the generalized assignment problem : An optimizing and heuristic approach. *INFORMS Journal on Computing*, 15(3):249–266, 2003.

[39] Y. Nesterov. Smooth minimization of non-smooth functions. *Mathematical Programming*, 103(1):127–152, 2005.

[40] M. Palpant, C. Artigues et C. Oliva. Mars : a hybrid scheme based on resolution search and constraint programming for constraint satisfaction problems. Rapport technique, LIA Avignon, 2004.

[41] A. Pigatti, M.P. de Aragao et E. Uchoa. Stabilized branch-and-cut-and-price for the generalized assignment problem. *Electronic Notes in Discrete Mathematics*, 5:389–395, 2005.

[42] D. Pisinger. A minimal algorithm for the 0-1 knapsack problem. *Operations Research*, 45(5):758–767, 1997.

[43] M. Posta, J.A. Ferland et P. Michelon. An exact method with variable fixing for solving the generalized assignment problem. *Computational Optimization and Applications*, 2011.

[44] M. Posta, J.A. Ferland et P. Michelon. Generalized resolution search. *Discrete Optimization*, 8(2):215–228, 2011.

[45] M.G.C. Resende et R.F. Werneck. A hybrid multistart heuristic for the uncapacitated facility location problem. *European Journal of Operational Research*, 174(1):54–68, 2006. ISSN 0377-2217.

[46] G.T. Ross et R.M. Soland. A branch and bound algorithm for the generalized assignment problem. *Mathematical programming*, 8(1):91–103, 1975.

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

[48] M. Sun. Solving the uncapacitated facility location problem using tabu search. *Computers and operations research*, 33(9):2563–2589, 2006. ISSN 0305-0548.

[49] L.A. Wolsey. *Integer programming*. Wiley New York, 1998.

[50] M. Yagiura, T. Ibaraki et F. Glover. An ejection chain approach for the generalized assignment problem. *INFORMS Journal on Computing*, 16(2): 133–151, 2004.

[51] M. Yagiura, T. Ibaraki et F. Glover. A path relinking approach with ejection chains for the generalized assignment problem. *European journal of operational research*, 169(2):548–569, 2006.