Mathematical programming methods for complex cutting problems

Quentin Viaud

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Quentin Viaud**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : MATHEMATIQUES APPLIQUEES ET CALCUL SCIENTIFIQUE

Mathematical programming methods for complex cutting problems

---

Date de soutenance : 11 décembre 2018

Devant la commission d'examen composée de :

Olivier BEAUMONT	Directeur de recherche, INRIA Bordeaux	Président du jury
Manuel IORI	Professeur, UNIMORE	Rapporteur
Ivana LJUBIC	Professeur, ESSEC Business School	Rapporteur
Sandra U. NGUEVEU	Maître de Conférence, Toulouse INP	Examinateur
Eric PINSON	Professeur, Université catholique de l'Ouest	Examinateur
François CLAUTIAUX	Professeur, Université de Bordeaux	Directeur
Ruslan SADYKOV	Chargé de recherche, INRIA Bordeaux	Co-Directeur
François VANDERBECK	Professeur, Université de Bordeaux	Co-Directeur

– 2018 –

Title: Mathematical programming methods for complex cutting problems

Abstract: This thesis deals with a two-dimensional bin-packing problem with defects on bins from the glass industry. Cutting patterns have to be exact 4-stage guillotine and items defect-free. A standard way to solve it is to use Dantzig-Wolfe reformulation with column generation and branch-and-price. This is impossible in our case due to large instance size. We first study and solve the defect-free pricing problem with an incremental labelling algorithm based on a dynamic program (DP), represented as a flow problem in a hypergraph. Our method is generic for guillotine knapsack problems but fails to solve large instance in a short amount of time. Instead we solve the defect-free bin-packing problem with a DP and a diving heuristic. This DP generates non-proper columns, cutting patterns that cannot be in an integer solution. We adapt standard diving heuristic to this "non-proper" case while keeping its effectiveness. We then extend the diving heuristic to deal with defects. Our first proposal heuristically repairs a given defect-free solution. Secondly the defect-free diving heuristic is adjusted to handle defects during column fixing. Our industrial results outline the effectiveness of our methods.

Keywords: Decomposition, Column generation, Hypergraph, Labelling algorithm, Diving heuristic, Cutting

---

Titre: Méthodes de programmation mathématiques pour des problèmes complexes de découpe

Résumé: Cette thèse s'intéresse à un problème de bin-packing en deux dimensions avec des défauts sur les bins rencontré dans l'industrie verrière. Les plans de découpe sont guillotine 4-stage exact, les objets à couper sans défauts. Une possible résolution utilise la décomposition de Dantzig-Wolfe puis une génération de colonnes et un branch-and-price. Cela est impossible dans notre cas du fait d'instances de trop grande taille. Nous résolvons d'abord le problème de pricing sans défauts par un algorithme incrémental de labelling basé sur un programme dynamique (DP), représenté par un problème de flot dans un hypergraphe. Notre méthode est générique pour les problèmes de sac-à-dos guillotine mais ne résout pas de larges instances en un temps de calcul raisonnable. Nous résolvons alors le problème de bin-packing sans défauts grâce à un DP et une heuristique de diving. Le DP génère des colonnes "non-propres", ne pouvant pas participer à une solution entière. Nous adaptons le diving pour ce cas sans perte d'efficacité. Nous l'étendons alors au cas avec défauts. Nous réparons d'abord heuristiquement une solution du problème sans défauts. La fixation des colonnes dans le diving sans-défaut est ensuite modifiée pour gérer les défauts. Les résultats industriels valident nos méthodes.

Mots-clés: Décomposition, Génération de colonnes, Hypergraphe, Algorithme de labelling, Heuristique de diving, Découpe

---

Unité de recherche: Institut de Mathématiques de Bordeaux - 351 cours de la Libération - 33 405 TALENCE - FRANCE

Titre: Méthodes de programmation mathématiques pour des problèmes complexes de découpe

Résumé long:

Introduction

Le cadre de cette thèse est de proposer des méthodes efficaces de résolution pour un problème industriel de découpe de verre. Ce dernier survient dans les lignes de production de fenêtre à double-vitrage. Le problème en lui-même est de concevoir un ensemble de plans de découpe pour des grandes plaques rectangulaires en verre et ce afin de découper un ensemble de pièces rectangulaires d'après un carnet de commandes donné. Ces pièces sont ensuite utilisées dans la fabrication de fenêtres. Ce problème s'inscrit dans la famille des problèmes d'optimisation de type découpe et placement. Dans ces derniers, le but est d'utiliser efficacement un ensemble de contenants afin de maximiser leur utilisation. Ces contenants peuvent être des rouleaux, des plaques rectangulaires, des fûts, des caisses ... Mieux utiliser ces contenants, les remplir du mieux possible, permet de maximiser le coût de découpe/de conditionnement. Dans cette thèse, ceci se traduit par la réduction du nombre de plaques de verre, ce qui permet de minimiser les chutes et la consommation de ressources. Cette considération est très importante surtout quand le matériau a un taux de production faible ou une haute valeur ajoutée. La plupart des problèmes de découpe et placement sont difficiles à résoudre à la fois d'un point de vue pratique et théorique. Les éléments qui rendent notre problème difficile mais stimulant sont la grande quantité de pièces, les contraintes propres à la découpe du verre et la présence de défauts dans les plaques de verre à considérer.

La production de fenêtre double-vitrage est divisée en deux grandes étapes: celle de découpe et celle de production. La première vise à découper un ensemble de grandes plaques rectangulaires en pièces rectangulaires de plus petite taille. L'étape de production utilise cet ensemble de pièces et les assemble pour fabriquer une fenêtre à double-vitrage. Dans cette thèse, nous nous intéressons au processus de découpe. Le fonctionnement standard est de recevoir un ensemble de plaques de verre. Chaque plaque est caractérisée par une longueur et hauteur. La norme sur les plaques est de six mètres pour la longueur et de trois mètres pour la hauteur. La qualité des plaques de verre produites est variable, et dépend du processus de fabrication, et du transport. Du fait de la complexité du processus de fabrication du verre, des bulles d'air peuvent apparaître. Le transport et la manipulation du verre depuis l'usine de production jusqu'à l'atelier de découpe peut conduire à l'apparition de microfissures. Ces bulles et fissures sont considérées comme des défauts. De manière formelle un défaut est approximé par un rectangle dont les coordonnées, la longueur et la hauteur sont connues. Quand les plaques de verre sont reçues en usine, et pour faciliter leur manipulation, elles sont stockées et empilées sur un support.

La dernière plaque empilée sera donc la première à être découpée. Afin de respecter les commandes de ses clients, l'usine doit découper différentes pièces utilisées dans la fabrication de futures fenêtres. Chaque pièce de verre est rectangulaire de longueur et hauteur données. Puisque le problème traité ici est lié à la production de double-vitrage, la demande de chaque rectangle de verre est d'environ deux.

A partir d'une commande client et d'un ensemble de plaques, le processus de découpe est le suivant. Initialement, les plaques sont empilées près de l'atelier de découpe. La première plaque disponible est dépilée et placée sur une table de découpe. Un traçage est ensuite réalisé sur la plaque à l'aide d'une molette. Ce marquage se décompose de plusieurs lignes droites traversant la plaque d'un bord à l'autre et parallèle aux autres bords. Ces lignes forment un plan de découpe. Après la phase de traçage, une pression est appliquée de chaque côté d'un trait de découpe pour séparer le verre en deux sous-plaques. Cette opération est appelée une coupe ou découpe guillotine. Par application récursive de coupes guillotines le long des traits de découpe, la plaque initiale est découpée en plaques plus petites. Celles-ci sont de deux types : soit une pièce à produire, soit un déchet. Les pièces sont ensuite stockées, les déchets sont jetés. Tant que toutes les pièces ne sont pas produites, une nouvelle plaque est dépilée et le processus de découpe recommence.

Il y a bien entendu des contraintes sur la manière de découper le verre. La principale est que n'importe quelle plaque ne peut être découpée que par une découpe dite guillotine. Le trait de découpe marque la plaque et l'opération de rompage permet de scinder un morceau de verre en deux morceaux de plus petites tailles. Cette façon de découper est obligatoire car découper d'une autre manière peut entrainer un risque de création de fissure dans la plaque et conduirait à devoir la jeter entièrement. Le nombre de découpes à appliquer pour obtenir une pièce peut être arbitrairement grand. Cependant plus ce nombre est grand, plus il est complexe et long d'obtenir les pièces. Afin de limiter la complexité des plans de découpe produits, une limite de quatre coupes est fixée avant d'obtenir une pièce. Pour respecter un cahier des charges, chaque pièce découpée doit d'être sans défaut. Il est également interdit de découper à travers un défaut. Ceci conduit les plans de découpe à devoir localiser les défauts dans les chutes de verre. Il est toutefois autorisé d'effectuer une rotation à 90° des pièces. D'un point de vue pratique, l'ordre d'extraction des pièces est réalisé de gauche à droite et de bas en haut

Une solution valide à notre problème est de concevoir un ensemble de plans de découpe respectant les contraintes de découpe des plaques et les contraintes d'ordre entre les plaques. Notre objectif est de trouver un ensemble de plans de découpe garantissant la plus faible perte de matière première. Pour une solution donnée, sa qualité est mesurée en sommant la longueur des plaques dans lesquelles au moins une pièce est découpée, et en soustrayant la longueur du résiduel sur la dernière plaque, qu'on appelle longueur résiduelle. Plus cette

dernière est longue, plus il est facile de pouvoir y redécouper des pièces. De la manière dont les pièces sont extraites, ceci tend à positionner les pièces de la dernière plaque le plus à gauche possible.

Le travail de la thèse est décomposé en quatre parties. La première introduit les définitions des problèmes de découpe et placement. La seconde se concentre sur la manière de concevoir un plan de découpe de bonne qualité pour une plaque. La troisième détaille comment résoudre notre problème en omettant les défauts. La dernière partie traitera du problème industriel tel quel en utilisant les résultats des parties précédentes.

L'objectif du travail est de proposer des méthodes efficaces de résolution pour notre problème. Les principaux enjeux sont d'être capables de traiter de grande quantité de pièces à découper, les défauts présents sur les plaques et ce en un temps de calcul raisonnable.

Etat de l'art

Avant de proposer de nouvelles méthodologies pour résoudre notre problème, nous proposons un tour d'horizon des techniques utilisées dans la littérature. Les problèmes de découpe et placement sont très souvent liés à des applications industrielles spécifiques. Du fait de cette grande diversité et richesse des problèmes de découpe, il convient de définir une façon de les nommer. Deux problèmes principaux sont traités dans ce document : le problème de sac-à-dos guillotine en deux dimensions et le problème de bin-packing en deux dimensions. Le premier consiste à trouver un multiensemble de pièces rectangulaires de valeur donnée maximum qui peuvent être découpées dans une plaque rectangulaire donnée et tel que deux pièces ne se chevauchent pas. Pour obtenir les pièces, la propriété de découpe guillotine doit également être satisfaite. Le second problème minimise le nombre de plaques à utiliser pour découper un ensemble de pièces. Nous considérons ici la variante de ces problèmes avec défauts.

Une approche classique de résolution des problèmes de bin-packing par programmation mathématique est d'utiliser la reformulation de Dantzig-Wolfe. Le problème initial de bin-packing est décomposé en deux problèmes : le problème maître et le problème de pricing. Le problème maître décide des plans de découpe à choisir pour chaque plaque, le second a pour but de les générer et de gérer les contraintes de découpe. Une façon de résoudre les problèmes obtenus après reformulation est d'utiliser la génération de colonnes pour le problème maître et de résoudre le problème de pricing par programmation dynamique. Si les plaques sont supposés être différentes les unes des autres, chacune possède son propre problème de pricing. Une solution réalisable est ensuite obtenue par branch-and-price ou heuristique de diving. Cette reformulation et manière de résoudre le problème ont fait leurs preuves sur le problème de bin-packing à une dimension. De plus, l'application de la décomposition de Dantzig-Wolfe

à ce problème met en évidence une formulation pseudo-polynomiale pour le problème de pricing. Par réécriture il est possible d'obtenir une formulation pseudo-polynomiale pour le problème initial. Ce genre d'approche peut ensuite être étendu au problème en deux dimensions. Cependant, le principal facteur limitant est la taille de la formulation. En pratique, il est difficile de résoudre des problèmes de découpe considérant beaucoup de niveaux de découpe mais des approches ont réussi à gérer les problèmes en deux et trois niveaux de coupe. La principale limitation reste cependant la taille des instances des problèmes traités. Pour un passage à plus grande échelle, des méthodes heuristiques basées sur le problème reformulé sont préférées. Les plus efficaces sont les heuristiques dérivées de la génération de colonnes. Elles ont l'avantage de s'appuyer sur l'information captée durant cette dernière. Cet ensemble de méthodes donnent des résultats satisfaisants pour les problèmes de découpe en deux dimensions mais sans défauts sur les plaques. Quand ces derniers sont considérés, le problème devient beaucoup plus difficile et des heuristiques sont souvent utilisées.

Au final, la reformulation de Dantzig-Wolfe se révèle très efficace pour des problèmes où les pièces à découper sont peu nombreuses mais de quantité à produire élevée. Dans notre cas, la quantité est faible et les dimensions des pièces sont hétérogènes. Une utilisation directe des méthodes de l'état de l'art se révèle donc difficile. Ajoutons à cela la difficulté induite par la présence de défauts. Nous pouvons toutefois réécrire notre problème industriel en utilisant la reformulation de Dantzig-Wolfe. La principale difficulté se situe alors dans le problème de pricing, de type sac-à-dos guillotine en deux dimensions avec défauts. De prime abord, il semble difficile de résoudre exactement notre problème par branch-and-price. Toutefois, des heuristiques de diving pourraient nous permettre d'obtenir des solutions de bonne qualité.

L'approche retenue pour résoudre notre problème industriel se décompose en deux étapes : résoudre le problème sans défaut puis étendre les résultats obtenus au problème avec défauts. Dans le cas du problème sans défauts, nous nous focalisons d'abord sur la résolution du problème de pricing obtenu après reformulation de Dantzig-Wolfe. En effet, afin de générer des colonnes rapidement il est primordial de mettre au point une résolution efficace de ce dernier.

Problème de sac-à-dos guillotine en deux dimensions sans défauts

Notre problème de pricing est un problème de sac-à-dos guillotine en deux dimensions. Nous proposons d'abord de résoudre un problème simplifié en relâchant les contraintes de production maximum sur les pièces. Nous autorisons donc à surproduire certaines pièces. Ce problème relâché, dit "unbounded", est résolu grâce à un programme dynamique ayant une représentation par hypergraphe. Un sommet de l'hypergraphe représente un état du

programme dynamique, un hyperarc représente une transition entre plusieurs états. Afin de réduire la taille de l'hypergraphe et donc du programme dynamique, nous développons plusieurs techniques de simplification de ce dernier. Les principales utilisent des règles de dominance sur la manière de découper une plaque. L'une des plus efficaces est de supprimer les symétries. Une seconde est de renforcer les contraintes de production des pièces directement dans la structure de l'hypergraphe. Ceci est réalisé par énumération partielle de certains plans de découpe. Une dernière technique est d'effectuer un filtrage par coûts Lagrangiens, développé pour le Problème de Plus Court Chemin Élémentaire avec Ressources, et ici étendu aux hypergraphes. L'idée sous-jacente est de supprimer des hyperarcs ne pouvant pas intervenir dans une solution optimale du problème de pricing. Une fois l'hypergraphe simplifié, le programme dynamique associé est résolu. Bien entendu ce dernier résout une relaxation de notre problème de pricing initial. Afin d'obtenir une solution réalisable, nous proposons de réécrire le programme dynamique comme un problème de flot dans l'hypergraphe. En y intégrant des contraintes annexes pour borner la quantité de pièces à produire, cela donne lieu à un modèle mathématique. Ce dernier est ensuite résolu par un solver de programmation linéaire en nombre entiers. Cette façon de résoudre le problème est limitée par le nombre de variables entières qui dépend de la taille de l'hypergraphe. En pratique, elle ne permet donc pas de résoudre des problèmes de grande taille. Pour ce faire, nous développons un algorithme de labelling. En théorie il est possible d'écrire un programme dynamique qui assure d'obtenir un plan de découpe valide sans surproduction de pièces. En pratique implémenter ce programme se révèle impossible du fait de sa taille exponentielle. Notre idée est de démarrer avec le programme dynamique "unbounded". Une fois résolu, deux cas sont possibles : soit la solution de ce programme dynamique respecte les contraintes de production et donc la solution est optimale, soit il existe au moins une pièce qui est surproduite. Quand le second cas se présente, nous proposons d'intégrer une nouvelle dimension à notre programme dynamique. L'idée est d'assurer que la pièce surproduite précédemment ne le soit plus. Ceci fait cependant croitre la taille du programme dynamique initial. Notre proposition est d'ajouter itérativement des dimensions. Pour essayer de contenir l'explosion du nombre d'états de nos programmes dynamiques étendus, nous intégrons le filtrage sur coût réduit Lagrangien et développons des règles de dominances entre des solutions partielles.

La méthodologie proposée par nos travaux sur le problème de sac-à-dos guillotine en deux dimensions peut être utilisée dans de nombreuses variantes de ce problème. Nous testons nos algorithmes sur des données de la littérature et des données industrielles. Nos expérimentations numériques portent premièrement sur les règles de simplification de l'hypergraphe. Dans un second temps, nous testons trois approches de résolution exacte : la résolution de notre problème comme un problème de flot de cout maximum avec con-

traintes annexes pour assurer le respect des contraintes de production ; un algorithme de labelling par extension de l'espace d'état et filtrage sur coût réduit Lagrangien ; une variante de notre algorithme de labelling avec une règle de dominance forte entre les états. Les premiers résultats expérimentaux valident l'impact des techniques de réduction de la taille de l'hypergraphe, notamment celle reposant sur l'énumération de plans de découpe partiels. Nos algorithmes de labelling ont une bonne efficacité pour résoudre des problèmes de taille industrielle.

Problème de bin-packing en deux dimensions sans défauts

De notre méthode exacte pour résoudre le problème de pricing, nous nous focalisons ensuite sur le problème industriel de bin-packing en deux dimensions. Dans un premier temps, nous travaillons sur une relaxation en ne considérant pas les défauts présents dans les plaques. L'avantage est qu'il n'y a plus qu'un seul problème de pricing à considérer puisque les plaques sont identiques. Les défauts seront ensuite gérés en post-traitement. Nous nous intéressons alors au problème de bin-packing en deux dimensions sans défauts.

De la reformulation de Dantzig-Wolfe, le problème de pricing à résoudre est un problème de sac-à-dos guillotine en deux dimensions. Bien que nous ayons un algorithme de résolution exacte pour ce problème, ce dernier peut parfois être couteux en temps de calcul. Faire converger la génération de colonnes avec cet algorithme pourrait se révéler extrêmement long. Cependant, résoudre le programme dynamique "unbounded" associé au problème de pricing se résout de manière très rapide. Afin d'obtenir une méthode de résolution performante, notre proposition est de résoudre le problème de pricing grâce au programme dynamique "unbounded" au lieu de le résoudre exactement. La conséquence directe est que, dans la plupart des cas, l'oracle de pricing va renvoyer une colonne non réalisable vis-à-vis des contraintes de production des pièces. En effet, le programme dynamique autorise la surproduction. Ces colonnes non réalisables sont dites non-propres. Notre idée est de forcer leurs intégrations dans la formulation du maître et procéder au déroulement classique de la génération de colonnes. Nous remarquons que puisque des colonnes non-propres sont insérées dans le problème maître, une dégradation de la qualité de la relaxation linéaire du maître est à prévoir après convergence. De nos expérimentations, cette dégradation est peu importante mais le gain en termes de temps de calcul est toutefois non négligeable.

Une fois la convergence atteinte et afin d'obtenir une solution réalisable, nous proposons d'appliquer une heuristique de diving. Le principe de cette heuristique est de choisir une colonne et de lui attribuer une valeur entière, de mettre à jour le problème maître associé et de refaire une génération de colonnes. Le fonctionnement s'apparente à faire une recherche sur une branche de l'arbre de branch-and-price. Il existe une particularité à prendre en compte

en voulant fixer une colonne. Puisque que notre problème de pricing est résolu de manière relâché, la solution fractionnaire du maître peut contenir des colonnes non-propres. Toutefois fixer une colonne non-propre n'est pas autorisé puisque que non réalisable vis-à-vis des contraintes du problème de pricing. Le choix d'une colonne à fixer se fait donc en deux étapes. La première est de choisir une colonne propre dans la solution fractionnaire du maître et la fixer. Si aucune n'a pu être fixée, la seconde étape construit heuristiquement des colonnes propres. Ceci nous garantit de toujours fixer une colonne propre et de créer une solution partielle réalisable. Après fixation la génération de colonnes non-propres redémarre. Ce procédé de fixation est utilisé tant que le maître restreint est réalisable. Durant le processus de fixation, nous gardons la meilleure solution entière. Afin d'améliorer nos chances de trouver une solution de bonne qualité, nous utilisons une heuristique dite de complétion. Cette heuristique démarre après chaque fixation et construit une solution réalisable à partir de la solution partielle déjà fixée. Pour diversifier nos solutions, nous réalisons plusieurs explorations de l'arbre de branch-and-price en changeant les colonnes fixées. Pour accélérer notre méthode, nous utilisons également la stabilisation dans la génération de colonnes et autorisons à générer plusieurs colonnes à chaque itération.

Une originalité de notre travail est de résoudre le problème avec une heuristique de diving et ce en générant des colonnes non-propres. Cette simplification dans la résolution du problème de pricing le rend beaucoup plus rapide à résoudre. D'un point de vue global, la qualité de la solution finale est excellente. Nous avons testé notre méthode en changeant le nombre de « dive » dans l'arbre de branch-and-price et la façon dont nous créons des solutions heuristiques. Nos expérimentations sur des instances de la littérature et industrielles montrent l'efficacité de nos approches. La méthodologie est également générique et peut être réutilisée sur d'autres problèmes de découpe.

Problème de bin-packing en deux dimensions avec défauts

Dans la dernière partie de notre travail, nous résolvons notre problème industriel de découpe. A partir des observations faites précédemment, il est possible d'utiliser la reformulation de Dantzig-Wolfe et de le résoudre par génération de colonnes en utilisant une heuristique de diving. En pratique, cela implique de pouvoir écrire un programme dynamique "unbounded" pour chaque problème de pricing et ce en prenant en compte les défauts associés à chaque plaque. Ceci se révèle actuellement difficile à faire car cela impliquerait de pouvoir écrire un tel programme sans que l'espace d'états ne soit de taille trop importante. Ensuite, il faudrait être capable de stocker en mémoire un programme dynamique pour chaque problème de pricing. Pour parer à cela, nous développons deux alternatives. La première est de ne pas considérer les défauts et de les traiter en post-traitement. La résolution s'effectue par l'heuristique de

diving et des heuristiques de réparation. La seconde est de prendre en considération directement les défauts dans l'heuristique de diving.

Notre première approche est de résoudre le problème sans défauts par l'heuristique de diving. Les défauts sont ensuite considérés en post-traitement par des heuristiques de réparation. La méthode principale sur laquelle reposent les heuristiques de réparation est un algorithme de permutation. De la contrainte de coupe guillotine, un plan de découpe peut se représenter par un arbre. Cette représentation est utilisée par l'algorithme de permutation. De la superposition entre un plan de découpe et une plaque, le but est de trouver une permutation de ce plan tel que les défauts se retrouvent dans les chutes. Puisque qu'il n'est pas toujours possible d'éliminer totalement les défauts entre un plan et une plaque, l'algorithme de permutation est modifié pour garantir à minima que l'aire totale des pièces chevauchant un défaut est minimale. Ceci implique de devoir redécouper une ou des pièces mais à moindre coût. Comme mentionné précédemment, cette routine de permutation est ensuite intégrée dans une heuristique de réparation. La première consiste à prendre chaque plan de découpe et chaque plaque à la suite. Une version plus évoluée utilise une représentation par graphe biparti entre les plans de découpe et les plaques. Le but est de trouver un couplage de coût minimum. Le poids d'un arc de ce graphe est la perte engendrée par l'affectation d'un plan de découpe à une plaque. Puisqu'il existe un risque de chevauchement entre une pièce et un défaut, et ce même après application de la routine de permutation, nous avons développé des heuristiques de réparation de manière à être capable de reconnaître ce cas et de simplement redécouper la pièce dans une autre plaque.

Cette première approche pour le problème avec défaut se révèle capable de résoudre des instances industrielles. Néanmoins elle est considérée comme "myope" car l'information sur les défauts, connue a priori, n'est utilisée qu'a posteriori. Afin de tirer avantage de cette dernière, nous modifions l'heuristique de diving pour le problème sans défauts. Le problème de pricing est résolu comme précédemment. La modification apportée se situe sur la manière dont les colonnes sont fixées. Le processus de fixation se décompose en trois étapes. La première sélectionne les colonnes propres dans la solution fractionnaire du maître. Pour une colonne propre, la routine de permutation est utilisée pour savoir la plaque à laquelle elle sera affectée. Dans le cas où il n'existe aucune colonne propre, diverses heuristiques constructives sont utilisées. La colonne et la plaque sélectionnées sont ensuite fixées et ce, en prenant en compte la contrainte d'ordre entre les plaques.

Nous réalisons divers tests de sensibilité de nos méthodes en faisant varier le nombre de défauts par plaques. Dans le cas industriel, ici traité, nos approches sont compétitives l'une avec l'autre. La différence d'efficacité dépend du nombre de défauts présents dans chaque plaque. Quand ce dernier est petit, il est en fait assez simple de pouvoir éliminer les défauts en les positionnant dans des chutes. Quand le nombre de défauts par plaque est élevé, la version

modifiée de l'heuristique de diving est à préférer.

Conclusion

Pour conclure, les travaux réalisés dans cette thèse ont permis de proposer des méthodes pour résoudre un problème industriel de découpe de verre. Les principales difficultés rencontrées se situaient dans le gestion des défauts sur les plaques, cas peu traité dans l'industrie, des contraintes de découpe à considérer ainsi que la volumétrie des problèmes à résoudre. Nous avons proposé une approche de résolution basée sur la génération de colonnes pour résoudre le problème sans défauts. Cependant, nous nous sommes heurtés à une difficulté supplémentaire relative au problème de pricing de type sac-à-dos guillotine en deux dimensions. Ce manuscrit traite de la résolution exacte de ce problème de pricing, de la résolution par heuristique de diving pour le problème général de découpe avec et sans défauts.

Le problème de pricing reste un problème difficile à résoudre surtout pour des instances de grande taille. Pour ce dernier, nous avons développé une méthode basée sur une reformulation par réseau et flot. Celle-ci est obtenue par réécriture d'un programme dynamique représentant l'ensemble des plans de découpe valides pour une plaque mais autorisant la surproduction de pièces. Par ajout de contraintes supplémentaires pour limiter la production de pièces, le modèle de flot permet de résoudre le problème de pricing. Ceci est cependant limité par la taille du modèle. Pour dépasser cette limitation, nous avons développé et combiné le filtrage sur cout réduit Lagrangien, une stratégie incrémentale d'augmentation de l'espace d'état du programme dynamique ainsi qu'un algorithme de labelling. Pour obtenir un hypergraphe de taille réduite, nous avons aussi utilisé des techniques de simplifications basées sur la structure des plans de découpe à produire. Nos expérimentations numériques soulignent l'impact de nos méthodes sur des instances de la littérature et industrielles.

A partir des résultats sur le problème de pricing, nous avons développé une méthode heuristique de résolution pour le problème de bin-packing sans défauts. De résultats préliminaires, il a été observé que résoudre une version relâchée du problème de pricing par programmation dynamique détériore peu la qualité de la relaxation linéaire du maitre après convergence comparé à résoudre exactement le problème de pricing. Ce faisant, nous avons implémenté la génération de colonnes en autorisant la création de colonnes non-propres. Celles-ci ont la particularité de ne pas respecter les contraintes du problème de pricing, la surproduction de pièces étant autorisée. Afin d'obtenir une solution entière, nous avons développé une heuristique de diving gérant cette spécificité. Itérativement une colonne propre est sélectionnée et fixée. Si aucune n'existe, nous en créons une heuristiquement. Notre implémentation se révèle capable de traiter des instances de grande taille en un temps de calcul raisonnable.

Des bons résultats de notre heuristique de diving pour le problème sans

défauts, nous l'avons étendue à notre problème industriel. Deux approches ont été retenues. La première se base sur une résolution du problème de découpe sans prise en compte des défauts et une opération de post-traitement pour réparer la solution. Ce dernier utilise un algorithme de permutation basé sur la structure des plans de découpe. Cette approche par réparation est néanmoins restreinte. En effet, elle possède une forte dépendance à la solution initiale du problème sans défauts pour construire une solution valide pour le problème avec défauts. Pour prévenir ce comportement, nous avons modifié notre heuristique pour directement gérer les défauts sur les plaques durant le processus de fixation de colonne. Nos expérimentations sur des instances réelles valident nos approches et soulignent des résultats exploitables industriellement.

Pour conclure, nous avons proposé des techniques avancées basées sur la génération de colonnes et la programmation dynamique pour résoudre un problème industriel de découpe de verre. D'un point de vue industriel, il serait intéressant d'étudier des variantes du problème avec par exemple des contraintes de date échue sur la production des pièces ou améliorer la conception des lots de pièces à découper.

Mots-clés: Décomposition, Génération de colonnes, Hypergraphe, Algorithme de labelling, Heuristique de diving, Découpe

S'il n'y a pas de solution, c'est qu'il n'y a pas de problème.

Acknowledgments

I first would like to express my sincere gratitude to François Clautiaux, Ruslan Sadykov and François Vanderbeck for having given me the opportunity to work on this research subject. I am grateful for their advice, remarks and their support during the thesis.

Besides my advisers, I thank Ivana Ljubic and Manuel Iori to have accepted to be my rapporteurs and also Sandra U. Ngueveu, Eric Pinson and Olivier Beaumont to have been members of my jury.

I am also grateful to Jorge E. Mendoza, Eric Pinson and David Rivreau who taught me operations research and who inspired me to continue to work in this field.

I thank all members of REALOPT, SVI and OMM teams to have warmly welcomed me. I will remember the good working atmosphere, the stimulating discussions around cups of coffee and all the fun we had during these last years.

My thoughts are also dedicated to my friends, those who were there before I started and also the ones I encountered during my thesis.

I finally would like to thank my family for their support.

Contents

Contents **xix**

Introduction **1**

1 State of the art **5**
1.1 Definition and notations 5
1.1.1 Bin-packing problems 6
1.1.2 Knapsack problems 7
1.1.3 Specific variants of bin-packing problems 10
1.2 Generic solution methods for cutting and packing problems . . . 13
1.2.1 Initial formulation of 1BP 13
1.2.2 Pattern based formulation 14
1.2.2.1 Dantzig-Wolfe decomposition 14
1.2.2.2 Column generation 15
1.2.2.3 Application of Dantzig-Wolfe decomposition to 1BP 15
1.2.2.4 Solving the pricing problem 16
1.2.2.5 Branching schemes 17
1.2.3 Dynamic programming and pseudo-polynomial formulations 18
1.2.3.1 Max-cost flow pricing problem formulation . . . 18
1.2.3.2 Pseudo-polynomial size formulation for the 1BP 19
1.2.4 Classical heuristics for cutting and packing problems . . 20
1.2.4.1 Basic rounding heuristics 20
1.2.4.2 Pure diving heuristic 21
1.2.4.3 Diving heuristic with limited backtracking . . . 22
1.3 Two-dimensional knapsack problem 24
1.3.1 Problem instances 24
1.3.2 Dynamic programming for the unbounded 2KP 25
1.3.3 Exact methods for the bounded 2KP 28
1.3.3.1 ILP based exact methods for two-stage 2KP . . 28
1.3.3.2 ILP based exact methods for three-stage 2KP . 30
1.3.3.3 ILP based exact methods for any-stage 2KP . . 32

1.3.3.4 Recursive exact algorithm for any-stage 2KP . . 33
1.3.3.5 Branch-and-bound approaches 35
1.3.3.6 Constraint programming 36
1.3.4 Heuristic approaches 37
1.3.4.1 Bottom-up heuristics 37
1.3.4.2 Top-down heuristics 38
1.3.4.3 Primal-dual heuristics 39
1.4 Two-dimensional bin-packing problem 40
1.4.1 Problem instances 41
1.4.2 Pseudo-polynomial size ILP formulations 42
1.4.3 Heuristic approaches 43
1.4.3.1 One-phase heuristics 43
1.4.3.2 Two-phase heuristics 44
1.4.3.3 Meta-heuristics 45
1.4.3.4 ILP based heuristics 46
1.5 Variants of two-dimensional cutting problems 47
1.5.1 Two-dimensional cutting problems with leftover 48
1.5.2 Two-dimensional cutting problems with defects 50
1.6 Conclusion 50

2 Knapsack problem **53**
2.1 A dynamic program for the unbounded 2KP 53
2.1.1 Dynamic program for the U-2KP-RE-4-r 54
2.1.2 Hypergraph representation of the dynamic program . . . 55
2.1.3 Hypergraph preprocessing 57
2.1.3.1 Simple pattern enumeration 57
2.1.3.2 Symmetry breaking 58
2.1.3.3 Simple plate reduction 59
2.1.3.4 Enhanced plate reduction 60
2.1.3.5 Hypergraph simplification 61
2.1.3.6 Enhanced pattern enumeration 62
2.2 A direct ILP formulation for the bounded 2KP 66
2.3 Lagrangian filtering 67
2.3.1 Standard resource constrained longest path problem . . . 68
2.3.2 Extension to the case of a hypergraph 69
2.3.3 Optimizing Lagrangian multipliers 71
2.3.3.1 Using a subgradient algorithm 72
2.3.3.2 Using column generation 72
2.3.3.3 Using row-and-column generation 73
2.4 A label setting algorithm for the bounded 2KP 74
2.4.1 Dynamic program for the bounded case 74
2.4.2 Forward labelling in the extended space 76
2.4.3 Decremental State Space Relaxation 78

2.4.4 Iterative labelling algorithm . . . 80
2.5 Heuristics for the bounded 2KP . . . 85
2.5.1 Hypergraph based heuristic . . . 85
2.5.2 Evolutionary heuristic . . . 86
2.6 Computational experiments . . . 88
2.6.1 Computational experiments for basic hypergraph simplifications . . . 89
2.6.1.1 Hypergraph simplifications . . . 89
2.6.1.2 Lagrangian cost filtering procedure . . . 90
2.6.1.3 Exact methods . . . 92
2.6.2 Computational experiments for partial pattern enumeration . . . 95
2.6.2.1 Hypergraph simplifications . . . 95
2.6.2.2 Lagrangian cost filtering procedure . . . 96
2.6.2.3 Exact methods . . . 98
2.7 Conclusion . . . 102

3 Bin-packing problems 103
3.1 Solving the 2BP$_l$ . . . 103
3.1.1 Formulations of the 2BP$_l$ . . . 103
3.1.1.1 Standard formulation . . . 104
3.1.1.2 Pseudo-polynomial size formulation . . . 105
3.1.2 Diving heuristic with non-proper columns . . . 106
3.1.2.1 "Non-proper" diving heuristic . . . 106
3.1.2.2 Completion heuristic . . . 109
3.1.3 Initial bounds . . . 110
3.1.3.1 Initial primal bounds . . . 110
3.1.3.2 Initial dual bounds . . . 111
3.1.4 Computational experiments . . . 111
3.1.4.1 Impact of the partial enumeration . . . 112
3.1.4.2 Comparison of exact methods and heuristics . . . 114
3.2 Solving consecutive 2BP and 2BP$_l$ . . . 118
3.2.1 Problem description . . . 118
3.2.2 Problem formulations . . . 120
3.2.3 Modified diving heuristic . . . 122
3.2.4 Diving heuristic for the C-2BP$_l$ . . . 123
3.2.5 Computational experiments . . . 123
3.3 Conclusion . . . 125

4 Bin-packing problems with defects 127
4.1 Problem formulation . . . 127
4.2 Post-processing methods . . . 129
4.2.1 Naive reparation . . . 130

4.2.2 Reparation by subplates permutation 131
4.2.3 Reparation by solving the 1BP 135
4.3 Diving heuristic with non-proper columns 136
4.3.1 "Non-proper" diving heuristic 137
4.3.2 Column fixing 138
4.3.3 Completion heuristic 140
4.4 Computational experiments 141
4.4.1 Impact of post-processing methods 142
4.4.2 Impact of modified diving heuristic 143
4.5 Conclusion 144

Conclusion **145**

Acknowledgments **149**

Bibliography **151**

Introduction

The scope of this thesis is to propose efficient solution methods for an industrial glass cutting problem, which arises in a double-paned windows production line. The problem is to design cutting patterns for large rectangular glass plates in order to cut a set of rectangular pieces, which are used to produce windows. The problem discussed here belongs to the class of cutting and packing problems. In these problems, the aim is to use effectively some initial inputs and to reduce raw material losses. Finding good quality cutting patterns is important since it contributes to reduce waste. Savings are even higher when one considers raw material of high value or with a slow production rate. Most packing problems are theoretically and practically difficult to solve. In our problem, a large amount of pieces, the constraints related to the glass cutting process, and the presence of defects on the glass plates present a real challenge for state-of-the-art optimization algorithms.

The production process of double-paned windows is divided in two main steps: the cutting process and the production process. The cutting process cuts a set of large rectangular glass plates into smaller rectangular glass pieces. The assemblage step uses this set of glass pieces and transforms them into double-paned windows. In this thesis, we focus on the cutting part of the process.

The standard input of the factory is a set of large rectangular glass plates. Each glass plate is characterized by its width and height. The norm on plate size is 6 meters by 3 meters. Due to the complexity of the glass melting process, air bubbles may appear on glass plates. Plates are transported from a melting site to the cutting factory. The transportation and manipulation of glass plates may lead to the apparition of tiny cracks on them. Bubbles and cracks are considered as defects. Formally a defect is characterized by two coordinates in the plan, a width and a height. For convenience if a defect has a non rectangular form, it is approximated by drawing a minimum size rectangle around it. When glass plates are received at the cutting factory and to ease their manipulation, they are stored in a rack. This rack has the structure of a stack. The last inserted plate is the first one to be out.

To ensure orders of its customers, the factory has to cut different rectangular glass pieces used later to create double-paned windows. Each glass piece is characterized by its width and height. Since double-paned windows are produced in the factory, the expected demand of each glass piece is two.

Due to customer orders it may be higher or lower. From a practical point of view, each glass piece after the cutting process is stored in a container. This container is then used to create windows.

From a customer order and a set of glass plates, the cutting process is the following. The first available plate is taken from the glass plate stack and placed on a huge table. Then tracing is performed with a glass cutter to mark the plate. These marks are edge-to-edge straight lines parallel to the edges of the glass plate: they represent a cutting pattern. After the marking step, an operator applies pressure to both sides of a straight line created with the diamond. This propagates a crack in the glass along the line and divides the glass plate in two smaller plates. This process is called a *guillotine cut*. By recursively applying cuts along marked lines, an initial glass plate is cut into smaller plates. Each of them is either a glass piece or a waste. When all glass pieces and wastes are obtained after cutting a glass plate, a new one is retrieved from the stack. This process is repeated as long as there are glass pieces to cut.

There are constraints on how to cut the glass plates. The first one is that only guillotine cuts can be performed on a glass plate. This cutting constraint is mandatory for the glass. Performing non-guillotine cuts creates cracks spreading through the whole glass plate. To simplify handling operations, there is a limit on the number of possible cuts to obtain a glass piece. This limit is now set to four, i.e. each glass piece has to be obtained using at most four guillotine cuts. It is obviously forbidden to cut trough a defect and glass pieces have to be defect free. Consequently when a cutting pattern is applied on a plate, defects of this plate have to be contained in wasted glass parts. It is possible to make a 90° rotation of glass pieces. When a cutting pattern is applied to a glass plate and due to guillotine cuts, pieces are always extracted in a bottom-left fashion.

The overall goal of solving this cutting problem is to obtain good quality solutions defined as follows. A solution is a set of cutting patterns. For a given solution, its quality is measured by summing up the width of plates in which a cutting pattern is applied minus the leftover of the last plate. This metric is related to the fact that a long leftover in the last plate can be reused for the next batch.

This document is decomposed in four parts. The first one introduces notation and definitions of cutting and packing problems. The second part focuses on how to design a good quality pattern for a glass plate. The third part details how to solve the industrial glass cutting problem while ignoring defects. The last part explains how to solve the industrial glass cutting problem based on insights in the third part.

The main motivation to study this problem is that it is critical to have an efficient solving procedure to handle large production plans or orders in a short amount of time. An overview of solving methods for cutting and

packing problems is given in Chapter 1. An efficient way to solve them is to apply the Dantzig-Wolfe reformulation. The problem can be tackled using column generation and dynamic programming. A solution is then obtained using branch-and-price or diving heuristic. In many cases, this approach gives good results when many glass pieces of the same size have to be produced. In our case, glass pieces are very heterogeneous, *i.e.* demands for glass pieces of the same size are very small. Thus state-of-the-art methods are hardly applicable.

The pricing problem in the above mentioned column generation approach is the two-dimensional guillotine knapsack problem. The state-of-the-art methods to solve it are based on reformulation techniques and compact formulations. In Chapter 2, new efficient exact methods to solve large scale instances of this problem are proposed. They are based on a dynamic program to handle glass cutting constraints. It can be represented by a hypergraph. Nevertheless, this dynamic program does not handle glass piece production bounds. In practice overproduction is not possible since it is forbidden to store glass pieces in the factory. To overcome this limitation, the dynamic program is written as a flow problem and enriched with side-constraints. Such flow problem may be tackled with labelling algorithms in the case of a simple graph. An originality of the work presented here is to extend this methodology to hypergraphs. To speed up our algorithms, we use preprocessing rules and hyper-arc elimination by Lagrangian costs. A partial enumeration of the dynamic program is also proposed to enforce item production bounds. This results in a method that is able to solve the two-dimensional guillotine knapsack problem in a reasonable amount of time. However when the problem has to be solved repeatedly in the context of column generation, it is too time consuming to solve exactly.

In a column generation method for the two-dimensional guillotine cutting stock problem, two issues arise: the pricing problem is too hard to be solved exactly at each iteration, and the number of nodes in a branch-and-price tree may be large, since the number of possible patterns, and the number of constraints are numerous. Ways to overcome these issues are discussed in Chapter 3. To obtain a feasible solution, *i.e.* a set of cutting patterns, a diving heuristic is used instead of a full branch-and-price method. One key ingredient of our diving algorithm is that it first solves a relaxed subproblem in which glass piece overproduction is possible. The motivation is that such patterns can be obtained efficiently using dynamic programming. Since in practice it is forbidden to overproduce a glass piece, the classical diving heuristic has to be adapted to produce a feasible solution. This is achieved by iteratively selecting a suitable pattern in the solution produced by column generation or, if such pattern does not exist, heuristically creating a new one. The diving heuristic is also hybridized with an evolutionary heuristic to increase its performance. Reported results prove that the approach is efficient even for large scale problems. In the same time, a study is done to measure the performance of our diving heuristic

for a production day in the cutting factory.

In practice, the defect-free assumption may not match industrial constraints. Since the diving heuristic obtains results of good quality, we propose to generalize it to handle defects on plates in Chapter 4. Two extensions are proposed. The first one is based on post-processing techniques. The idea is to solve the problem without defect management and then to adjust the patterns produced to repair the solution. The second proposed method is to integrate the defect management directly in the diving heuristic, *i.e.* to assign generated patterns to glass plates with defects each time a pattern is fixed. Different ways to fix patterns are described and experimentally compared to each other.

Chapter 1

State of the art

This chapter describes the current state of the art on cutting problems. This literature review is used as a starting point for the work described in this manuscript. The first part of this chapter details the most common variant of cutting problems and notations used through the remainder of this manuscript. These explanations are mandatory to well define the different variants of cutting problems. Indeed, they are numerous since cutting problems are related to many different industrial applications. Two main problems are analyzed here: the two-dimensional knapsack problem and the two-dimensional bin-packing problem. The first is related to find a multiset of rectangular pieces of maximum given value which can be cut in a given rectangular plate such that cut pieces do not overlap. The two-dimensional bin-packing problem aims to minimize the number of rectangular plates required to pack a subset of rectangular pieces. The second part of this chapter details the commonly used solving procedure for cutting problems based on Dantzig-Wolfe reformulation and solved with column generation. Extra formulations and advanced techniques for cutting problems are also detailed. Explanations are supported with a standard cutting problem. Next parts of the chapter focus on solving methods for the two-dimensional knapsack problem and the two-dimensional bin-packing problem. Exact and efficient solving methods are described for both problems taking into account industrial constraints. Heuristic approaches are also detailed. Finally a description of two-dimensional bin-packing problem variants is given.

1.1 Definition and notations

Cutting and packing problems are common problems in combinatorial optimization and have many variants and restrictions due to industrial requirements. The purpose of this section is to define notations that will be used through this manuscript. First, the general definition of bin-packing problem is given. Second, the same definition is given for the knapsack problem. This

section concludes by a description of bin-packing problem variants. Note that a typology of cutting and packing problem already exists (see Wäscher et al. [87]). The notation explained hereinafter does not used standard notations defined in this typology since the latter is heavy and problems discussed here are very specific.

1.1.1 Bin-packing problems

Bin-packing problems are combinatorial optimization problems and have a practical importance since they arise as industrial problems in different contexts. Main applications are in the cutting industry of raw materials like paper, wood, steel and glass. The purpose of this section is to describe classical bin-packing problems.

The bin-packing problem (1BP) is an optimization problem which aims to minimize the number of bins required to pack a subset of items of given width. A problem instance is represented by a pair $D = (\mathcal{I}, W)$, where $\mathcal{I}$ is the set of items to pack and W is the bin capacity. Bins are assumed identical and in sufficient quantity to cut all items. Each item i has a fixed width w_i and has to be cut exactly d_i times. A packing for a given bin B is said to be valid if the sum of the width of items cut in this bin does not exceed the bin capacity. It is standard to consider that input data are integer. A standard application is to cut items with different widths from rolls of fixed width. A solution representation is depicted in Figure 1.1.

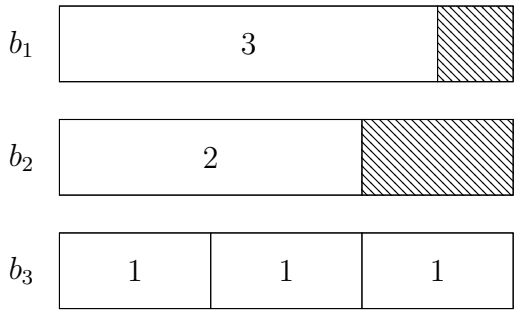

Figure 1.1: Solution representation for a 1BP instance using three bins to cut three times item 1, one time item 2 and one time item 3. Dashed lines represent waste parts of bins.

The two-dimensional bin-packing problem (2BP) is the generalization of the 1BP. It aims to minimize the number of rectangular containers (bins) required to pack a subset of rectangular items of given dimensions. A problem instance is represented by a pair $D = (\mathcal{I}, B)$, where $\mathcal{I}$ is the set of rectangular items to pack and B is the standard rectangular bin to use. Bins are assumed identical and in sufficient quantity to cut all items. Each item i has a fixed width and height (w, h) and has to be packed exactly d_i times. Each bin has a

fixed width and height (W, H). A packing for a given bin is said to be feasible if items packed in it have their edges parallel to the bin edges, no items are out of the bin and two distinct item copies do not overlap. It is standard to consider that input data are integer. A solution to a 2BP instance is often represented by a set of packings, one packing for one bin. An example of such solution is given in Figure 1.2. A standard industrial application is to cut rectangular planks to build furniture or to cut glass to manufacture windows. In that case, the term cutting pattern (or pattern) is used instead of packing.

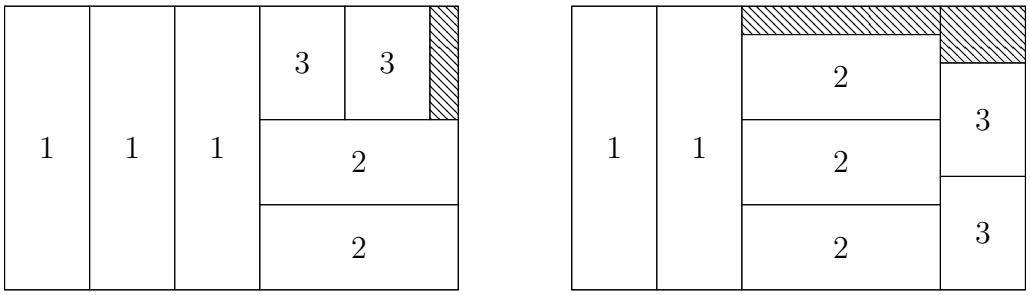

Figure 1.2: Solution representation for a 2BP instance using two bins to cut five times item 1, five times item 2 and four times item 3. Dashed lines represent waste parts of bins.

The main difference between the 1BP and 2BP is related to the following feasibility problem. Given a multiset of items $\tilde{\mathcal{I}} \subseteq \mathcal{I}$ packed in a bin B, is there a feasible configuration to pack items $\tilde{\mathcal{I}}$ in bin B ? Remark that the multiset $\tilde{\mathcal{I}}$ may contain several copies of an initial item $i \in \mathcal{I}$. This feasibility problem is trivial for the 1BP. It requires to check if the sum of the width of packed items does not exceed the bin capacity and if for each item $i \in \mathcal{I}$, the number of copies of i contained in $\tilde{\mathcal{I}}$ does not exceed its availability d_i. For the 2BP the feasibility problem is NP-complete.

1.1.2 Knapsack problems

There is a wide range of variants for packing problems. Most of them have specific requirements due to different industrial machines used by companies. In the following part of this section, a generic description is given for the one and two-dimensional knapsack problems. Details are then given for standard industrial requirements related to two-dimensional knapsack problems.

The one-dimensional integer knapsack problem (1KP) is the problem of finding an item multiset of maximum value without exceeding a given capacity. A problem instance is a pair $D = (\mathcal{I}, W)$, where $\mathcal{I}$ is the set of items to pack and W is the bin capacity. Each item i has a fixed width w_i, a given profit e_i and can be selected at most d_i times. The 1KP is sometimes called bounded one-dimensional knapsack problem. The notation 1KP is preferred instead. A

special variant of this problem is the binary one-dimensional knapsack problem ($1KP_b$) in which an item can be selected at most one time.

The two-dimensional knapsack problem (2KP) is the generalization of the 1KP. The problem is to find a multiset of rectangular items of maximum value which can be packed in a given rectangular bin such that packed items do not overlap. A problem instance is represented by a pair $D = (\mathcal{I}, B)$, where $\mathcal{I}$ is the set of rectangular items to pack and B is the bin. Each item i has a fixed width and height (w_i, h_i), a given profit e_i and can be packed at most d_i times. The bin B has fixed width and height (W, H).

As mentioned previously, 2KP is often considered to match industrial requirements especially in the cutting industry. The main related variant is the guillotine cut property. From a practical point of view, a guillotine cut is an edge-to-edge cut parallel to the edges of the bin. When cutting a plate or a bin with such cuts, this always divides it into two subplates. Figure 1.3 shows an example of guillotine and non guillotine cutting patterns. In Figure 1.3(a), the pattern follows guillotine property because it is possible to cut the items with vertical and then horizontal cuts. In Figure 1.3(b), items cannot be cut using guillotine cuts only. Advantage of guillotine cuts is that they make handling easier and they do not degrades quality of the raw material. Indeed a non guillotine cut on some raw material such as glass can lead to cracks on the plate. This results in raw material losses, which cost money for a company.

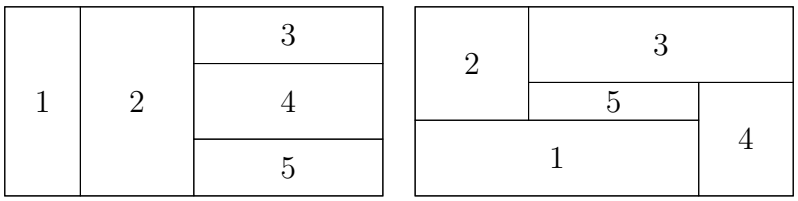

Figure 1.3: Guillotine (a) and non guillotine (b) cutting patterns

In addition to guillotine cuts, another possible requirement is to guarantee that items are obtained after a certain number of recursive cuts. A cutting pattern is often said to be staged when the number of stages (*i.e.* the number of guillotine cuts) allowed to cut an item is bounded by an integer value. The most common problems in the literature are two-stage and three-stage. The two-stage restriction requires to cut each item in at most two guillotine cuts, the three-stage restriction requires to cut each item in at most three guillotine cuts, *etc.* A problem with k cutting stages is simply called k-stage. When the number of stages is infinite, the problem is called any-stage. The direction of first stage cuts may be either horizontal or vertical and the cuts of the same stage have to be in the same direction. The cut directions of any two adjacent stages must be perpendicular to each other. Figure 1.4 shows an example of

two and three-stage patterns. In Figure 1.4(a), items from 1 to 3 are produced by horizontal and then vertical cuts (red and blue lines). Items 4 and 5 are cut in the same way. In Figure 1.4(b), items from 1 to 4 are produced as in Figure 1.4(a). Items 5 and 6 are cut by using one horizontal cut (red line), one vertical cut (blue line) and then one horizontal cut (green line).

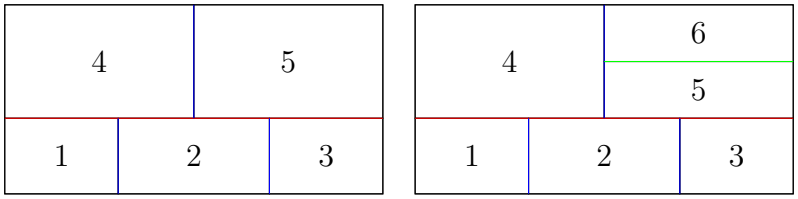

Figure 1.4: Two-stage (a) and three-stage (b) guillotine cutting patterns

A less standard cut property occurs at the last stage of a k-stage problem. If an additional cut is allowed only to separate an item from a waste area, the problem is said to be with trimming or *non-exact*. If such extra cut is not allowed the problem is called *exact*. An example is given in Figure 1.5 for a two-stage pattern. Exact case pattern is depicted in Figure 1.5(a). Figure 1.5(b) is non-exact since an extra cut (trimming) has to be performed to obtain items 2 and 5.

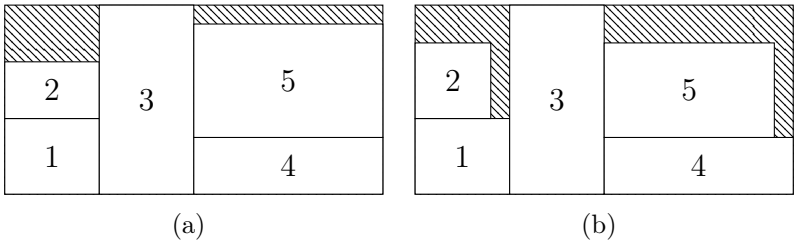

Figure 1.5: Exact (a) and non-exact (b) 2-stage guillotine cutting patterns

An additional cutting constraint is to restrict the set of possible cut lengths. A cut is *restricted* if its length has to be equal to the width w_i or the height h_i of some item $i \in \mathcal{I}$. In the case of restricted cuts, one of the two produced subplates after such cut has to be initialized with an item of width or height equal to the cut length. This forces to immediately extract an item after such cut. Figure 1.6 shows an example of restricted and non-restricted patterns. Figure 1.6(a) is restricted because cuts are related to item dimensions. Figure 1.6(b) is non-restricted since items can only be obtained with cut length larger than item dimensions.

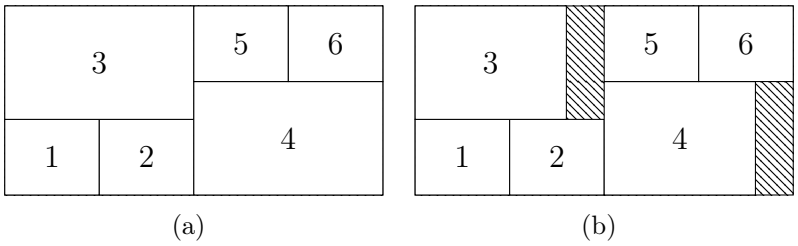

Figure 1.6: Restricted (a) and non-restricted (b) guillotine cutting patterns

Standard properties of cutting problems are to allow the rotation of each item $i \in \mathcal{I}$ and to bound the number of times each item is cut. The first one means that it is allowed to cut a duplicate (h_i, w_i) of an initial item (w_i, h_i). When there exists no upper bound on the number of times an item can be cut (*i.e.* $d_i = +\infty, \forall i \in \mathcal{I}$), the problem is said to be unbounded or unconstrained. The problem is called bounded or constrained otherwise.

Since the 2KP can accommodate several cutting constraints, the following notations are used in this document to characterize the different problem variants:

- C (resp. U) indicates that the demand of each item is bounded (resp. unbounded)
- NR means that cut lengths are non-restricted and non-exact, NRE that cuts are non-restricted and exact, R that cuts are restricted non-exact and RE means restricted exact
- k is an integer which corresponds to the maximum number of stages, while ∞ is associated to the any-stage variant
- f (resp. r) does not allow item rotation (resp. allow item rotation)

Unless otherwise stated the guillotine cut requirement is assumed to be always imposed. For instance, notation C-2KP-RE-4-f refers to the bounded restricted exact 4-stage problem without item rotation, notation U-2KP-R-∞-r is related to the unbounded restricted non-exact any-stage problem with item rotation.

1.1.3 Specific variants of bin-packing problems

Industrial requirements do not only restrict the way items are cut but also how to use bins. A description of some common variants of 1BP and 2BP is given hereinafter.

The most widespread variant of 2BP is the two-dimensional strip-packing problem (2SP). In such problem an item set $\mathcal{I}$ has to be packed in a bin (or

roll) B of fixed height and infinite width. The objective is to minimize the width of the bin. A problem instance is represented by a pair $D = (\mathcal{I}, B)$, where $\mathcal{I}$ is the set of rectangular items to pack and B is the single bin to use. Bin has an infinite width and a fixed height (∞, H). Each item i has a fixed width and height (w_i, h_i) and has to be packed exactly d_i times. Input data are considered integer. Figure 1.7 shows a 2SP solution representation.

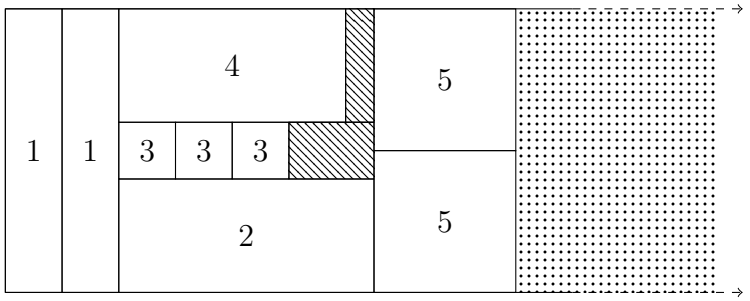

Figure 1.7: Solution representation for a 2SP instance using two bins to cut two times item 1, one time item 2, three times item 3, one time item 4 and two times item 5. Dashed lines represent waste parts of strip, dotted lines represent the unused part of the strip.

Other variants of the 2BP often modify the input data. A standard one is the multiple size two-dimensional bin-packing problem ($2BP_s$). The main difference with the 2BP is that bins are distinct from each other. A problem instance will be represented by a pair $D = (\mathcal{I}, \mathcal{B})$, where $\mathcal{I}$ is the set of rectangular items to pack and $\mathcal{B}$ is the set of available bins of given width and height (W_b, H_b). Some cases of the $2BP_s$ can limit the number of times a bin type is available. This leads to a maximum availability for each bin type.

Another variant of the 2BP considers defects on bins ($2BP_d$). Objective is the same as for the 2BP but bins are considered to be different. All bins $b \in \mathcal{B}$ have the same dimension (W, H) but are characterized by a specific set of defects $\mathcal{D}_b$. In practice, defects may not be rectangular but with guillotine cuts, only rectangular pieces will be removed anyway. Each defect $d \in \mathcal{D}_b$ is then defined by two coordinates (x_d, y_d) and a width and height (w_d, h_d). If a defect has a non rectangular form it is approximated by drawing a minimum size rectangle around it. If the defect set of a bin b is empty, the bin is defect free. A possible pattern for one bin is represented in Figure 1.8.

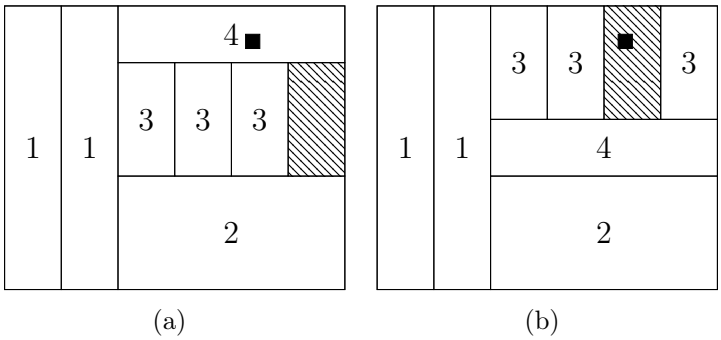

Figure 1.8: Infeasible (a) and feasible (b) cutting patterns for one bin with one defect. Item cut in pattern on the left overlaps a defect which is not permitted. Dashed lines represent waste parts of the bin, black box represents the defect.

The last variant of the 2BP is when the leftover of the last bin has to be also maximized ($2BP_l$). From a practical point of view such consideration is important since a large leftover can potentially be reused to pack extra items for a further command. The objective of the $2BP_l$ is hierarchical, first to minimize the number of bins to pack items as in the 2BP and second to minimize the used length in the last bin. This forces packed items in the last bin to be put to the left of it. Two equivalent solutions with and without leftover consideration are represented in Figure 1.9.

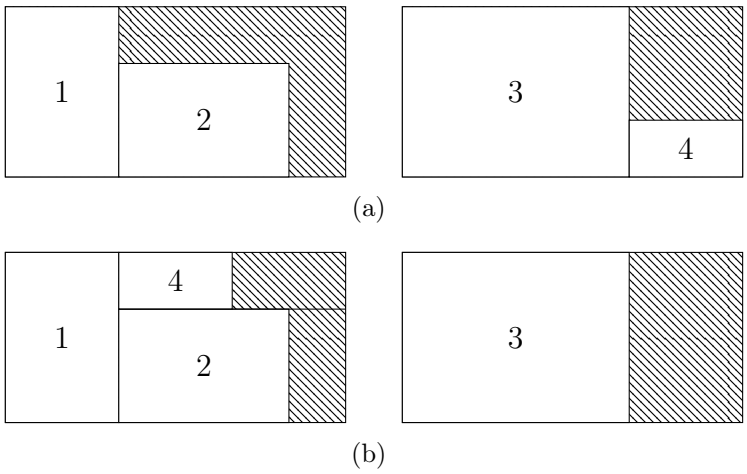

Figure 1.9: Without leftover (a) and with leftover (b) consideration for a 2BP instance. Solutions are equivalent for the 2BP since they both used two bins but solution (b) is preferred for $2BP_l$ because leftover part on the right of the second bin can be reuse more easily. Dashed lines represent waste parts of the bin.

Different 2BP variants can also be combined together. For instance, one may want to solve a multiple size two-dimensional bin-packing problem with defects and with leftover ($2BP_{sdl}$). Through this document notations related to defects and leftovers will be used also for the 2KP.

1.2 Generic solution methods for cutting and packing problems

The 1BP and 2BP can be decomposed in two problems. The first one is to assign items to bins and the other is to find feasible packings for each bin. A straightforward way to tackle these problems is to solve their integer linear programming (ILP) models by means of a commercial solver. However basic formulations have one of the two drawbacks: either its linear relaxation is weak (and thus the enumeration tree is huge) or its size is large and does not scale well raising memory consumption problems. In addition, symmetry problems occur. An approach to overcome these drawbacks is to exploit the problem structure using decomposition techniques to solve larger instances and obtain stronger formulations. The aim of this section is to explain the generic procedure to obtain the optimal solution of a generic ILP model using decomposition techniques. Explanations are supported with an application to solve 1BP, a detailed survey can be found in Delorme et al. [20].

1.2.1 Initial formulation of 1BP

This section introduces the most straightforward and historically first ILP formulation of 1BP by Kantorovich [46]. Let binary variable $y_b = 1$ if $b \in \mathcal{B}$ is used, 0 otherwise. Let also integer variables x_{ib} be the number of items $i \in \mathcal{I}$ cut from bin b. The formulation is:

$$\min \sum_{b \in \mathcal{B}} y_b \tag{1.1}$$

$$\text{s.t.} \sum_{b \in \mathcal{B}} x_{ib} = d_i, \quad \forall i \in \mathcal{I} \tag{1.2}$$

$$\sum_{i \in \mathcal{I}} w_i x_{ib} \leqslant W y_b, \quad \forall b \in \mathcal{B} \tag{1.3}$$

$$x_{ib} \in \mathbb{N}, \quad \forall i \in \mathcal{I}, \quad \forall b \in \mathcal{B} \tag{1.4}$$

$$y_b \in \{0, 1\}, \quad \forall b \in \mathcal{B} \tag{1.5}$$

Objective function (1.1) minimizes the number of used bins. Constraints (1.2) check that the requested number of items is cut. Constraint set (1.3) ensures that the total width of items cut in a bin do not exceed bin capacity.

This formulation has two main drawbacks as outlined by Martello and Toth [54]: poor quality of its linear relaxation and many symmetric solutions. Using a modern ILP solver some medium size instances can be solved to optimality. However, for large instances, the number of y_b and x_{ib} variables grows very quickly and even modern solvers may fail to solve them.

1.2.2 Pattern based formulation

Since the ILP formulation above may not be solved with modern solvers for large instances, other formulations have to be studied. This section describes a pattern based formulation for 1BP.

1.2.2.1 Dantzig-Wolfe decomposition

Dantzig-Wolfe decomposition is a powerful tool to reformulate ILP models in order to obtain a stronger linear programming (LP) relaxation. Assume a generic ILP of the form:

$$\min \mathbf{cx} \tag{1.6}$$
$$\text{s.t. } \mathbf{Ax} = \mathbf{b} \tag{1.7}$$
$$\mathbf{x} \in \mathbf{X} \tag{1.8}$$

with:

$$\mathbf{X} = \{\mathbf{x} \in \mathbb{Z}^n | \mathbf{Dx} \geqslant \mathbf{d}\} \tag{1.9}$$

The LP relaxation of this model may be of a poor quality. The idea of the reformulation is to exploit the structure of set $\mathbf{X}$. Assume that $\mathbf{X}$ is a non-empty bounded convex polyhedron. From Minkowski's theorem (see Nemhauser and Wolsey [61]), any point $\mathbf{x} \in \mathbf{X}$ can be expressed as a convex combination of its extreme points $\mathcal{P}$. Let $\mathbf{x}^p \in \mathbf{X}$ be a solution vector for each $p \in \mathcal{P}$, $\mathbf{X}$ can be expressed as follows:

$$\mathbf{X} = \left\{ \mathbf{x} = \sum_{p \in \mathcal{P}} \mathbf{x}^p \lambda_p | \sum_{p \in \mathcal{P}} \lambda_p = 1, \lambda_p \geqslant 0, \forall p \in \mathcal{P} \right\}$$

Dantzig-Wolfe decomposition, introduced by Dantzig and Wolfe [18], gives rise to the following reformulation of the ILP model (1.6)-(1.8), in which each variable λ_p corresponds to an extreme point $p \in \mathcal{P}$ of $\mathbf{X}$:

$$\min \sum_{p \in \mathcal{P}} (\mathbf{cx}^p) \lambda_p \tag{1.10}$$

$$\text{s.t.} \sum_{p\in\mathcal{P}}(\mathbf{A}\mathbf{x}^p)\lambda_p = \mathbf{b} \tag{1.11}$$

$$\sum_{p\in\mathcal{P}}\lambda_p = 1 \tag{1.12}$$

$$\lambda_p \in \mathbb{N}, \quad \forall p \in \mathcal{P} \tag{1.13}$$

1.2.2.2 Column generation

The number of variables λ is often exponential and the formulation cannot be solved directly. The usual way to solve it is to use column generation (see Gilmore and Gomory [33] and Desaulniers et al. [21]). The LP relaxation of (1.10)-(1.13) is called *master problem* (MP). Column generation aims to solve a so-called restricted master problem (RMP) containing only a subset of variables $\lambda_p, p \in \bar{\mathcal{P}}, \bar{\mathcal{P}} \subseteq \mathcal{P}$. The idea is to price variables λ_p (or columns) that are out of the formulation and insert them in the RMP to improve its objective value. Let π be the vector of dual variables related to (1.11), columns are priced by solving the following pricing problem:

$$\zeta(\pi) = \min\{(\mathbf{c} - \pi\mathbf{A})\mathbf{x} | \mathbf{x} \in \mathbf{X}\} \tag{1.14}$$

An optimal solution to the pricing problem (or subproblem) corresponds to a variable λ_p of reduced cost $(\mathbf{c} - \pi\mathbf{A})\mathbf{x}^p$. If it is negative, a new column λ_p is added to the RMP. If no new column cannot be added to the RMP, method stops. At the end of the process, the RMP is solved to optimality with only a subset of variables. If the RMP optimal solution is integer, process stops because an optimal integer solution to the initial problem is found. Nevertheless the obtained solution is usually fractional and therefore column generation is coupled with the branch-and-bound method, resulting in the branch-and-price approach. When the optimal solution is not integer, a branching is performed, which creates child nodes and adds additional constraints to their RMP. The LP relaxation of each child node is solved again by column generation. A generic branching scheme has been developed by Vanderbeck [84]. For most problems, specific branching schemes are preferred to achieve a better performance.

1.2.2.3 Application of Dantzig-Wolfe decomposition to 1BP

In order to obtain a stronger LP relaxation, ILP formulation for 1BP given by (1.1)-(1.5) is extended using Dantzig-Wolfe decomposition. Constraints (1.2) can be viewed as coupling constraints since they link items among all opened bins. Constraint set (1.3)-(1.5) is independent for each bin. Let $\mathcal{P}(b)$ be the set of feasible cutting patterns satisfying (1.3)-(1.5) for a given bin $b \in \mathcal{B}$. Let binary variable $\lambda_p^b = 1$ if bin $b \in \mathcal{B}$ uses pattern $p \in \mathcal{P}(b)$, 0 otherwise. Let

also a_{ip} be the number of times an item $i \in \mathcal{I}$ is cut from a given pattern p. According to Gilmore and Gomory [34] and Vance [81], the Dantzig-Wolfe reformulation of 1BP is the following:

$$\min \sum_{b \in \mathcal{B}} \sum_{p \in \mathcal{P}(b)} \lambda_p^b \tag{1.15}$$

$$\text{s.t.} \sum_{b \in \mathcal{B}} \sum_{p \in \mathcal{P}(b)} a_{ip} \lambda_p^b = d_i, \quad \forall i \in \mathcal{I} \tag{1.16}$$

$$\sum_{p \in \mathcal{P}(b)} \lambda_p^b \leqslant 1, \quad \forall b \in \mathcal{B} \tag{1.17}$$

$$\lambda_p^b \in \{0, 1\}, \quad \forall p \in \mathcal{P}(b), \quad \forall b \in \mathcal{B} \tag{1.18}$$

Objective function (1.15) aims to minimize the number of used bins (or patterns). Constraints (1.16) ensure to cut the required number of items. Constraint set (1.17) bounds the number of selected patterns for each bin to one at most.

Since there are at most $|\mathcal{B}|$ bins in the solution and one needs to price columns for each bin this leads to solve at most $|\mathcal{B}|$ pricing problems. However an instance of 1BP assumes that bins are identical. This implies that solving only one pricing problem is enough since bins are equivalent. This allows one to merge all pattern sets $\mathcal{P}(b), \forall b \in \mathcal{B}$ in only one pattern set $\mathcal{P}$. Thus, formulation (1.3)-(1.5) can be written in a more compact way, using substitution $\tilde{\lambda}_p = \sum_{b \in \mathcal{B}} \lambda_p^b$:

$$\min \sum_{p \in \mathcal{P}} \tilde{\lambda}_p \tag{1.19}$$

$$\text{s.t.} \sum_{p \in \mathcal{P}} a_{ip} \tilde{\lambda}_p = d_i, \quad \forall i \in \mathcal{I} \tag{1.20}$$

$$\tilde{\lambda}_p \in \mathbb{N}, \quad \forall p \in \mathcal{P} \tag{1.21}$$

Note that this time, variable $\tilde{\lambda}_p$ is the number of times pattern $p \in \mathcal{P}$ is used. As in the general Dantzig-Wolfe decomposition, linear relaxation of formulation (1.19)-(1.21), called master problem (MP), is tackled using column generation. Columns are generated by solving the pricing problem related to (1.19)-(1.21). Comparison of quality of lower bounds obtained by solving LP relaxations of several ILP formulation can be found in Valério de Carvalho [80].

1.2.2.4 Solving the pricing problem

In a column generation context to solve the RMP related to (1.19)-(1.21), new columns need to be generated. Let π_i be the dual price associated to each

demand constraint (1.20), the pricing problem used to find new columns takes the form:

$$\zeta(\pi) = \min \left\{ 1 - \sum_{i \in \mathcal{I}} \pi_i x_i | w_i x_i \leqslant W, x_i \in [0, d_i], x_i \in \mathbb{N}, \forall i \in \mathcal{I} \right\} \tag{1.22}$$

This pricing problem is an instance of the bounded knapsack problem 1KP in which one wants to select x_i times items $i \in \mathcal{I}$ such that profit of selected items π_i is maximized without exceeding bin width W and item availability d_i. A standard way to solve this 1KP is to transform it into the binary knapsack problem $1KP_b$ by creating d_i distinct copies of each item $i \in \mathcal{I}$. The number of distinct items is $g = \sum_{i \in \mathcal{I}} d_i$, $\mathcal{W} = [1, 2, \ldots, W]$ is the set of possible width position in the bin. An optimal solution to the $1KP_b$ can be obtained by dynamic programming using Bellman's equations:

$$U(i, w) = \max \left\{ \begin{array}{l} U(i-1, w), \\ U(i-1, w - w_i) + \pi_i \text{ if } w_i \leqslant w \end{array} \right\}, w \in \mathcal{W}, i > 0 \tag{1.23}$$

$$U(0, w) = 0 \tag{1.24}$$

In previous recurrence relations, each $U(i, w)$ is the maximum value which can be attained with a width less than or equal to w using first i items. The optimal solution value is then equal to $U(g, W)$. The solution of this knapsack problem defines a column with reduced cost $\zeta(\pi)$. If such cost is negative, the column is inserted in the RMP, otherwise no more attractive columns can be added to the RMP and the column generation stops.

The dynamic programming approach is a way to solve the $1KP_b$. To efficiently solve large problems, specialized methods are preferred instead, such as strengthened dynamic programming approach (see Martello et al. [53]) or branch-and-bound method (see Pisinger [64]).

1.2.2.5 Branching schemes

A way to branch on formulation (1.15)-(1.18) is to consider the following constraint set (see Vance [81]) :

$$\sum_{p \in \mathcal{P}(b)} a_{ip} \lambda_p^b \in \mathbb{N}, \quad \forall i \in \mathcal{I}, \quad \forall b \in \mathcal{B}$$

Let $\alpha_b = \sum_{p \in \mathcal{P}(b)} a_{ip} \lambda_p^b$ for a given $b \in \mathcal{B}$, a branching scheme is to separate on the α_b value:

$$\sum_{p \in \mathcal{P}(b)} a_{ip} \lambda_p^b \leqslant \lfloor \alpha_b \rfloor, \quad \sum_{p \in \mathcal{P}(b)} a_{ip} \lambda_p^b \geqslant \lceil \alpha_b \rceil$$

Such branching scheme is enforced by deleting columns that violate upper bound $\lfloor \alpha \rfloor$ on the first branch and lower bound $\lceil \alpha \rceil$ on the second branch. The updated pricing problem after branching is a knapsack problem with upper and lower bounds on item variables.

Another way to branch but this time on formulation (1.19)-(1.21) is to impose bounds on a column subset. Assume a given subset $\bar{\mathcal{P}} \subset \mathcal{P}$ and with $\alpha = \sum_{p \in \bar{\mathcal{P}}} \tilde{\lambda}_p$, branching occurs on :

$$\sum_{p \in \bar{\mathcal{P}}} \tilde{\lambda}_p \leqslant \lfloor \alpha \rfloor, \quad \sum_{p \in \bar{\mathcal{P}}} \tilde{\lambda}_p \geqslant \lceil \alpha \rceil$$

This second branching scheme was initially proposed by Vanderbeck [82]. Branching schemes presented here focus only on the 1BP. More details as well as techniques to increase branching efficiency can be found in Vance [81], Vanderbeck [82] and Vanderbeck [83].

1.2.3 Dynamic programming and pseudo-polynomial formulations

The branch-and-price method based on Dantzig-Wolfe decomposition allows one to solve the 1BP to optimality. Nevertheless without an efficient branching scheme and/or a pricing oracle, finding the optimal solution can take a large computation time. Another efficient model is based on an arc-flow formulation and was first proposed by Valério de Carvalho [79].

1.2.3.1 Max-cost flow pricing problem formulation

The pricing problem related to the 1BP is a bounded 1KP which can be solved using dynamic programming as outlined in Section 1.2.2.4. According to the paradigm of Martin et al. [56], the search of a maximum cost packing for a bin with the dynamic program (1.23)-(1.24) is equivalent to the search of a max-cost flow in the corresponding directed acyclic graph. The proposed dynamic program can therefore be represented by means of a graph. Let $G = (\mathcal{V}, \mathcal{A})$ be this graph. The vertex set $\mathcal{V}$ is composed of all positions between $\{0, 1, \ldots, W\}$. Let s (resp. t) be the source (resp. sink) of G corresponding to vertex 0 (resp. W). An arc is created between each pair of vertices v and w in $\mathcal{V}$ if their is an item $i \in \mathcal{I}$ such that $w_i = w - v$. Formally $\mathcal{A} = \{a | 0 \leqslant \mathcal{T}(a) < \mathcal{H}(a) \leqslant W \text{ and } \mathcal{H}(a) - \mathcal{T}(a) = w_i, \forall i \in \mathcal{I}\}$. $\mathcal{H}(a)$ (resp. $\mathcal{T}(a)$) defines the head (resp. tail) set of arc $a \in \mathcal{A}$. There also exist additional arcs between a vertex v and $v + 1, v \in \{0, 1, \ldots, W - 1\}$ to represent unoccupied portion of the bin. Let $\Gamma^+(v)$ be the set of successors of vertex $v \in \mathcal{V}$. and $\Gamma^-(v)$ be the set of predecessors of vertex $v \in \mathcal{V}$. A solution to the pricing

problem (*i.e.* a valid packing for a bin) is a path in graph G and the arcs in this path correspond to the items to pack in the bin.

Since the problem is then transformed to the longest path problem in a directed acyclic graph G, one can write the associated ILP model for this problem. Let π_i be the dual price associated to demand constraint (1.20). Let also binary variable $x_a = 1$ if the arc $a \in \mathcal{A}$ is in the path, 0 otherwise. The set of arcs that cover item $i \in \mathcal{I}$ is denoted by $\mathcal{A}(i)$. Once graph G is built, the pricing problem is formulated as the following longest path problem:

$$\max \sum_{i \in \mathcal{I}} \pi_i \sum_{a \in \mathcal{A}(i)} x_a \tag{1.25}$$

$$\text{s.t.} \sum_{a \in \Gamma^+(s)} x_a = 1 \tag{1.26}$$

$$\sum_{a \in \Gamma^-(v)} x_a - \sum_{a \in \Gamma^+(v)} x_a = 0, \quad \forall v \in \mathcal{V} \backslash \{s, t\} \tag{1.27}$$

$$\sum_{a \in \Gamma^-(t)} x_a = 1 \tag{1.28}$$

$$x_a \in \{0, 1\}, \quad \forall a \in \mathcal{A} \tag{1.29}$$

The total value of the path in (1.25) has to be maximized. Constraints (1.27) are related to flow conservation. Constraint (1.26) sends a flow of one unit from sources s, constraint (1.28) receives it at the sink t. Note that the proposed formulation has a pseudo-polynomial size related to the dynamic program size. Some arcs in the graph can be removed without loss of optimality by using preprocessing rules. Details of these rules can be found in Côté and Iori [14].

1.2.3.2 Pseudo-polynomial size formulation for the 1BP

According to Valério de Carvalho [79] and from the pseudo-polynomial size formulation for the 1KP, the 1BP can also be formulated as a min-cost flow problem. Starting from graph G defined in Section 1.2.3.1, let integer variables x_a be equal to the flow value going through arc $a \in \mathcal{A}$. The pseudo-polynomial size min-cost flow formulation of the 1BP is the following:

$$\min z \tag{1.30}$$

$$\text{s.t.} \sum_{a \in \Gamma^+(s)} x_a = z \tag{1.31}$$

$$\sum_{a \in \Gamma^-(v)} x_a - \sum_{a \in \Gamma^+(v)} x_a = 0, \quad \forall v \in \mathcal{V} \backslash \{s, t\} \tag{1.32}$$

$$\sum_{a \in \Gamma^-(t)} x_a = z \tag{1.33}$$

$$\sum_{a \in \mathcal{A}(i)} x_a = d_i, \quad \forall i \in \mathcal{I} \tag{1.34}$$

$$x_a \in \mathbb{N}^+, \quad \forall a \in \mathcal{A} \tag{1.35}$$

The total flow through the graph in (1.30) has to be minimized. This corresponds to minimize the number of bins to use. Constraints (1.31)-(1.33) are related to flow conservation. Constraints (1.34) ensure that each item demand is fulfilled.

The previous flow formulation can be used in to obtain an optimal integer solution to the 1BP. One just has to to write it directly within a commercial solver and get the optimal solution. Since the flow formulation is pseudo-polynomial, large problem instances can be computationally long to solve.

1.2.4 Classical heuristics for cutting and packing problems

The main focus of previous sections has been on how to solve the 1BP to optimality. It happens sometimes that obtaining an optimal solution can be time consuming or impossible due to the size of input data. In this case, constructive and/or based ILP heuristics (or matheuristics) are used to obtain good feasible solutions. Since the problems discussed in this manuscript are two-dimensional, specialized classical constructive heuristics for them will be discussed later. This section focuses on ILP based heuristics instead, on rounding and specific diving heuristics. A complete review and numerical experiments can be found in Sadykov et al. [72].

The motivation to use matheuristics is to exploit the tight lower bounds obtained by the Dantzig-Wolfe decomposition on one hand. On a second hand, they are useful to obtain good primal solutions for large scale problems for which exact methods are problematic to apply. Moreover a good primal solution can be used to avoid useless exploration of the branch-and-price tree if one wants to obtain an optimal integer solution.

An intuitive way to obtain a feasible integer solution is to exploit the structure of a fractional solution of the LP relaxation of the 1BP formulation (1.19)-(1.21). By rounding variables λ_p to integer values, one can build a partial solution. If a partial solution satisfies all master constraints, it defines a valid feasible solution.

1.2.4.1 Basic rounding heuristics

Since an integer solution is composed of integer variables λ_p, a fractional variable with a value close to an integer one is likely to be integer in an integer

solution. Assume an instance of 1BP in which item overproduction is allows. Under this assumption, by rounding λ_p variables to integer values, one can build a partial solution. This is equivalent to select cutting patterns and use them to form a feasible solution. However when item production cannot be exceeded, this rounding-up process does not guarantee to get a feasible integer solution. By rounding the wrong variables, this may results in infeasibility of the partial solution.

To avoid the rounding-up procedure drawback, it may be possible to obtain an integer solution from the MP after convergence using ILP. When column generation ends, the MP is composed of columns having fractional values. A technique to obtain an integer solution is to solve the ILP formulation of the MP after convergence by forcing all λ_p variables to have an integer value. Clearly this may produce an integer solution but only based on columns in the MP.

These two methods may be efficient for some optimization problems but they do not guarantee to obtain systematically a feasible integer solution. Their main limitation is that they both only use columns in the MP after convergence.

1.2.4.2 Pure diving heuristic

To avoid the limited scope of rounding heuristics, pure diving heuristic is preferred instead. This heuristic is based on the exploration of the branch-and-price tree in a depth first manner. The idea is to round fractional variables λ_p in the LP relaxation of the MP after convergence to integer values. This fixing procedure leads to obtain a smaller MP often called residual master problem. The LP relaxation of the residual MP is then solved by column generation as in branch-and-price. For instance the pure diving heuristic, used on the 1BP formulation (1.19)-(1.21), iteratively fix variables λ_p (*i.e.* select a cutting pattern to add in a partial solution).

Since the diving heuristic iteratively fixes a column, one can write it as a recursive function. At iteration $j > 0$, let $\mathcal{P}^{j-1}$ and d^{j-1} be the set of columns and the residual demand which defines the residual master problem. Let also $\bar{\lambda}$ be the partial solution and $\tilde{\lambda}$ the rounding of the master problem solution at iteration $j-1$. The generic procedure is given in Algorithm 1. To start the diving, one calls the procedure PURE-DIVING$(\mathcal{P}^0, d, \varnothing, \varnothing)$. The algorithm first starts by updating the residual master problem and the partial solution (lines 1-2). Then due to column fixing, the master is updated and only proper columns are kept (lines 3-4). A column is proper if it does not violate residual master problem constraints. Note that the master problem may become infeasible, which results in an early termination. The updated residual master is then solved using column generation and the associated solution λ^j is stored (lines 5-6). After solving the residual master problem,

the set of non integer columns $\mathcal{F}$ are stored among the set of valid columns $\mathcal{P}^j$ enriched with new columns obtained during residual master resolution. If set $\mathcal{F}$ is empty, this implies that all columns have an integer value and the algorithm stops. If this set is not empty, one needs to select a subset $\mathcal{R}$ of columns from $\mathcal{F}$ and then heuristically round their values (lines 7-10). This rounding operation is represented by notation $\lceil \lambda_p^j \rfloor$ which rounds a column $\lambda_p^j > 0$ to the least non-zero integer $\lceil \lambda_p^j \rceil$ greater than λ_p^j or to the greatest integer $\lfloor \lambda_p^j \rfloor$ less than λ_p^j. In the course of the algorithm, one looks for a primal feasible solution using the partial fixed solution $\bar{\lambda}$ and the integer columns in $\mathcal{P}^j \backslash \mathcal{F}$ (line 8). Finally the algorithm recursively calls itself to continue to fix columns (line 11).

Algorithm 1: PURE-DIVING($\mathcal{P}^{j-1}, d^{j-1}, \bar{\lambda}, \tilde{\lambda}$)

1 $d_i^j \leftarrow d_i^{j-1} - a_{ip}\tilde{\lambda}_p, \forall i \in \mathcal{I}$
2 $\bar{\lambda} \leftarrow \bar{\lambda} + \tilde{\lambda}$
3 preprocess the residual master problem by removing infeasible columns from $\mathcal{P}^{j-1}$
4 **if** *master problem is infeasible* **then** return let $\mathcal{P}^j$ be the set of remaining valid columns after preprocessing
5 solve the LP relaxation of the residual master problem with column generation
6 $\lambda^j \leftarrow$ solution of the LP relaxation of the residual master problem
7 let $\mathcal{F} = \left\{p \in \mathcal{P}^j : \lfloor \lambda_p^j \rfloor < \lambda_p^j < \lceil \lambda_p^j \rceil\right\}$
8 if $\bar{\lambda} + \{\lambda_p^j\}_{p \in \mathcal{P}^j \backslash \mathcal{F}}$ defines a primal solution, record it
9 $\tilde{\lambda}_p \leftarrow 0$
10 **if** $\mathcal{F} = \varnothing$ **then** return heuristically choose a column set $\mathcal{R} \subseteq \mathcal{F}$ and heuristically round their values $\tilde{\lambda_p} \leftarrow \lfloor \lambda_p^j \rceil$ such that $\tilde{\lambda_p} > 0, p \in \mathcal{R}$
11 call PURE-DIVING($\mathcal{P}^j, d^j, \bar{\lambda}, \tilde{\lambda}$)

1.2.4.3 Diving heuristic with limited backtracking

Since a pure diving heuristic is a depth-first search, a possible improvement is to use a limited backtracking procedure to perform multiple dives. This backtracking relies on a Limited Discrepancy Search (LDS). The idea is to set two parameters on the maximum depth $maxDepth$ and discrepancy $maxDisc$ to consider. Backtracking in the diving tree occurs up to $maxDepth$ and it is managed by using a tabu list to store forbidden branching decisions. Backtracking branches are considered as long as the number of branching decisions in the tabu list does not exceed $maxDisc$ parameter.

Diving with LDS is formalized in Algorithm 2. One first calls the procedure LDS-DIVING($\mathcal{P}^0, d^0, \varnothing, \varnothing\varnothing, 0$). The first steps of the algorithm are the same as in Algorithm 1. The main difference between the two algorithms is related

to the management of tabu list $\mathcal{Z}$ and depth level *depth*. This is handled by just selecting subset $\mathcal{R}$ of columns among the retained column set $\mathcal{F}$ but not in the tabu list $\mathcal{Z}$ (lines 3-4). Then the diving with LDS is called (line 5) and the tabu list is updated (line 6). The recursive calls to diving with LDS and updates of the tabu list are performed while the size of the tabu list does not exceed the maximum diving discrepancy $maxDisc$ and the current depth level *depth* does not exceed the maximum allowed depth $maxDepth$.

Algorithm 2: LDS-DIVING($\mathcal{P}^{j-1}, d^{j-1}, \bar{\lambda}, \tilde{\lambda}, \mathcal{Z}, depth$)

```
1 execute lines 1-8 from Algorithm 1
2 do
3 |   λ̃_p ← 0
4 |   if F\Z = ∅ then return heuristically choose a column set
  |   R ⊆ F\Z and heuristically round their values λ̃_p ← ⌊λ^j_p⌉ such that
  |   λ̃_p > 0, p ∈ R
5 |   LDS-DIVING(P^j, d^j, λ̄, λ̃, Z, depth + 1)
6 |   Z ← Z ∪ R
7 while |Z| ⩽ maxDisc and depth ⩽ maxDepth
```

The diving procedure with LDS is dependent on the maximum discrepancy and depth parameters. Clearly if both parameters are fixed to a huge value, the tree exploration takes more computation time. Nevertheless a deeper exploration may lead to better quality primal solutions. Consequently there is a trade-off between exploration and computation time. Figure 1.10 depicts trees with pure diving and diving with LDS.

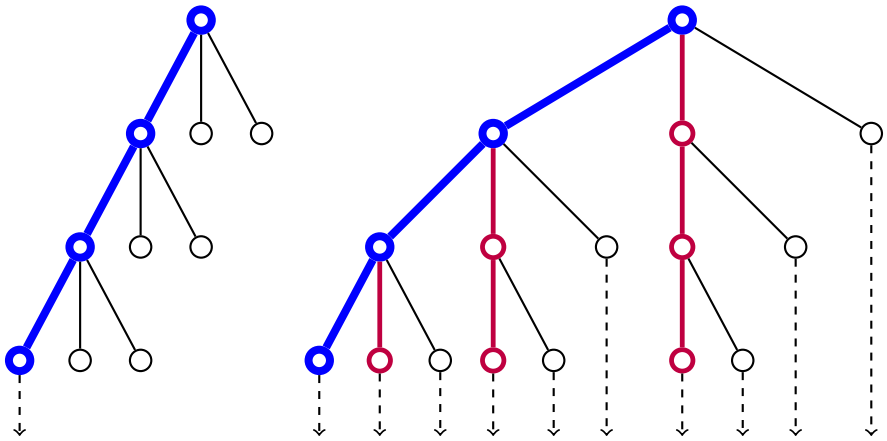

Figure 1.10: Example of pure diving (a) and diving with LDS ($maxDepth = 3, maxDisc = 2$)

1.3 Two-dimensional knapsack problem

The 2KP is the pricing problem used to create columns when solving the 2BP. Consequently one wants to design a method to solve it exactly and quickly. This problem is the subject of a large number of scientific papers. A description is first given to explain the structure of literature instances used to benchmark different approaches to solve 2KP. A focus is then done on dynamic programming for the unbounded 2KP followed by exact methods to solve the bounded 2KP. Finally some efficient heuristics are described. With respect to the problems discussed in this manuscript and if not mentioned, the guillotine cut requirement is assumed to be always used.

1.3.1 Problem instances

When a solving method is developed for an optimization problem, it is a common practice to test it on a set of problem instances. This set is often taken from industrial applications and reflects the consideration of companies. Using such data is twofold. On one hand it outlines the needs from companies to solve optimization problems they face off. On a second hand it allows researchers to benchmark their approaches and compare to each other.

The set of instances for the 2KP has grown with time. They are currently divided in four categories. The first one is related to constrained problem in which there exists a given item demand d_i. The second is for unconstrained problem where an item can be selected an unlimited number of time. Each of these two categories is then divided in two other subcategories. For some problem instances, the value e_i of each item is equal to its area $w_i \times h_i$. Such

instances are called unweighted in this case. When the profit of items is not equal to its area, the problem is called weighted. The retained notation to distinguish problem instances is the following. The notation CW (resp. CU) is used for constrained 2KP instances where the profit of items is weighted (resp. unweighted). For unconstrained 2KP instances, the notation UW (resp. UU) is used when profits are weighted (resp. unweighted).

The first contribution of 2KP instance has been done by Christofides and Whitlock [10], Wang [86] and Beasley [5]. In Christofides and Whitlock [10] and with collaboration to a wood cutting company, the authors made three CW instances available, named $CHW1-3$. The bigger one is $(W,H)=(40,70)$ and $|\mathcal{I}|=20$ with an average item demand of 3. Later Wang [86] adapts one of them and creates a CU instance instead, named W. To test its algorithms, Beasley [5] creates 13 UU instances, named $gcut1-13$. The bigger one is $(W,H)=(3000,3000)$ and $|\mathcal{I}|=32$. There also exist instances with smaller plate size $(W,H)=(1000,1000)$ but with higher number of items $|\mathcal{I}|=50$.

Another contribution to the number of available instances is done by Hifi [39]. The author defines CU instances derived from Christofides and Whitlock [10], named $1'$,$2'$ and $3'$. He also creates two new CW instances and three CU instances, named $A1-2$ and $A3-A5$. The biggest one has a size of $(W,H)=(132,100)$ and $|\mathcal{I}|=20$. He also adapts the UU instance of Herz [38] and modifies it to obtain a CU instance, named H. In a second time in Hifi [40], new large instances for the UU and UW problems are introduced, named $LU1-4$ and $LW1-4$. They can have dimension up to $(50.000,20.000)$ and $|\mathcal{I}|=200$.

For constrained problems, two small CU instances were proposed by Oliveira and Ferreira [62], named $OF1-2$, and five CW instances by Fekete and Schepers [28],named $okp1-5$. For them, the maximum bin size is $(100,100)$ and the number of items $|\mathcal{I}|=30$. In Fayard et al. [27], authors created 10 instances for both the CW and CU, named $CW1-11$ and $CU1-11$. The bin size is randomly generated between $[100,1000]$ and the number of items between $[25,60]$. Alvarez-Valdés et al. [2] also proposed new random CU and CW instances with size in range $[100,1000]$, named $APT30-39$ and $APT40-49$, Item sizes are in range $[0.05W,0.4W]$ and $[0.05H,0.4H]$, the demand of each item is set to be the minimum value between the geometric bound of the item $(\lfloor W/w_i\rfloor\lfloor H/h_i\rfloor)$ and a random integer number taken in range $[1,10]$. New CW and CU instances are also introduced in Cung et al. [17], named $CHL1-CHL4$,$Hchl1-2$,$Hchl9$ and $CHL1s-4s$,$CHL5-7$,$Hchl3s-8s$. The bin size is at most $(235,210)$ and the number of items is at most $|\mathcal{I}|=35$.

To summarize, the largest instances now available for the UW and UU are the ones from Hifi [41]. For the CW and CU variants, the set introduced in Alvarez-Valdés et al. [2] is a good one since bin are large enough. Remark that the set of constrained instances implies to work on small bin compare to unconstrained variants. It is common usage to transform unconstrained instances in

constrained one by setting a bound of one on each item. This gives larger instances. Most of described instances are available at `http://people.brunel.ac.uk/~mastjjb/jeb/orlib/cgcutinfo.html` and at `ftp://cermsem.univ-paris1.fr/pub/CERMSEM/hifi/2Dcutting/2Dcutting.html`.

1.3.2 Dynamic programming for the unbounded 2KP

As previously described the 1KP can be solved with a dynamic programming. A possible way to solve the 2KP is to extend this approach in two dimensions.

Pioneering work on the subject goes back to Gilmore and Gomory [34] where a first dynamic program for the U-2KP-NR-2-f was introduced. This one was latter extended by Beasley [5] to the k-stage variant. Assume a 2KP instance and let $F(k, w, h)$ (resp. $G(k, w, h)$) be the value of an optimal k-stage pattern related to a rectangle (w, h) where the first-stage cut direction is parallel to the w-axis (resp. to the h-axis). The set of possible length for any cuts parallel to the w-axis is $\mathcal{W} = [1, 2, \ldots, W-1]$ (resp. to the h-axis is $\mathcal{H} = [1, 2, \ldots, H-1]$). From Beasley [5], the generic recurrence relation for the U-2KP-NR-k-f is:

$$F(k, w, h) = \max \left\{ \begin{array}{l} F(0, w, h), \\ F(k, w, h') + F(k, w, h - h'), h' \in \mathcal{H}, \\ G(k-1, w, h) \end{array} \right\} \tag{1.36}$$

$$G(k, w, h) = \max \left\{ \begin{array}{l} G(0, w, h), \\ G(k, w', h) + G(k, w - w', h), w' \in \mathcal{W}, \\ F(k-1, w, h) \end{array} \right\} \tag{1.37}$$

$$G(0, w, h) = F(0, w, h) \tag{1.38}$$

$$F(0, w, h) = \max \{0, e_i | w_i \leqslant w, h_i \leqslant h, \forall i \in \mathcal{I}\} \tag{1.39}$$

Note that $F(0, w, h)$ represents the cut of an item and profit collection. The optimal solution to this dynamic program is obtained by computing the value $F(k', W, H)$ or $G(k', W, H)$ depending on the first cut direction. Clearly the main drawback of this dynamic program is its size. It becomes too large when the number of cutting stages and/or when the size of the bin increase as outlined by Gilmore and Gomory [34] and Beasley [5].

The state space size can be reduced without loss of optimality by using cutting properties. A straightforward way to do it is to reduce sets $\mathcal{W}$ and $\mathcal{H}$. From observations of Christofides and Whitlock [10], it is possible to move an item in a guillotine cutting pattern such that its bottom and left edges are adjacent to a cut or to the bin border. Remark also that position $w > W - \min_{i \in \mathcal{I}}\{w_i\}$ (resp. $h > H - \min_{i \in \mathcal{I}}\{h_i\}$) cannot have a piece lying to the right (resp. left) of the cut. Therefore a reachable position in a cutting pattern is expressed as a combination of item widths (resp. heights). This process is called pattern normalization. An example is given in Figure 1.11.

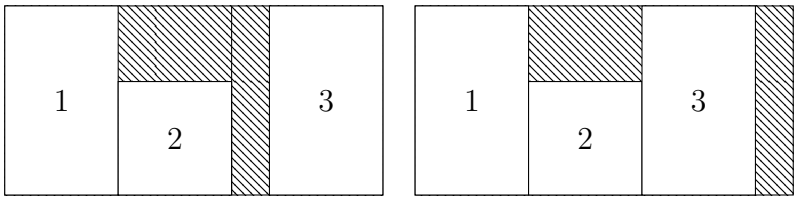

Figure 1.11: Example of a pattern without normalization (a) and with normalization (b)

Consequently one can rewrite sets $\mathcal{W}$ and $\mathcal{H}$ as follows:

$$\mathcal{W} = \left\{ w | w = \sum_{i \in \mathcal{I}} w_i a_i, 1 \leqslant w \leqslant W - \min_{i \in \mathcal{I}} \{w_i\}, a_i \geqslant 0, a_i \in \mathbb{N} \right\} \tag{1.40}$$

$$\mathcal{H} = \left\{ h | h = \sum_{i \in \mathcal{I}} h_i a_i, 1 \leqslant h \leqslant H - \min_{i \in \mathcal{I}} \{h_i\}, a_i \geqslant 0, a_i \in \mathbb{N} \right\} \tag{1.41}$$

Let functions $p(w)$ storing the width nearest from below to w in the new set $\mathcal{W}$ and $q(h)$ storing the height nearest to h in the new set $\mathcal{H}$:

$$r(w) = \max \{0, w' | w' \leqslant w, w' \in \mathcal{W}\}, w < W \tag{1.42}$$

$$q(h) = \max \{0, h' | h' \leqslant h, h' \in \mathcal{H}\}, h < H \tag{1.43}$$

The recurrence relations (1.36)-(1.39) are rewritten:

$$F(k, w, h) = \max \left\{ \begin{array}{l} F(0, w, h), \\ F(k, w, h') + F(k, w, q(h - h')), h' \in \mathcal{H}, \\ G(k - 1, w, h) \end{array} \right\} \tag{1.44}$$

$$G(k, w, h) = \max \left\{ \begin{array}{l} G(0, w, h), \\ G(k, w', h) + G(k, r(w - w'), h), w' \in \mathcal{W}, \\ F(k - 1, w, h) \end{array} \right\} \tag{1.45}$$

$$G(0, w, h) = F(0, w, h) \tag{1.46}$$

$$F(0, w, h) = \max \{0, e_i | w_i \leqslant w, h_i \leqslant h, \forall i \in \mathcal{I}\} \tag{1.47}$$

Since sets $\mathcal{W}$ and $\mathcal{H}$ are smaller, the dynamic program size becomes smaller.

In the same way, the any-stage problem can be modelled with a dynamic program. Let $F(-, w, h)$ be the optimal value of a pattern related to a rectangle (w, h). According to Beasley [5], the recurrence relation for the U-2KP-NR-∞-f is the following:

$$F(-,w,h) = \max \left\{ \begin{array}{l} F(0,w,h), \\ F(-,w,h') + F(-,w,q(h-h')), h' \in \mathcal{H}, \\ F(-,w',h) + F(-,r(w-w'),h), w' \in \mathcal{W} \end{array} \right\} \quad (1.48)$$

The dynamic program (1.48) has many symmetries. It is possible to remove some of them by imposing a restriction on cuts. This was first outlined by Christofides and Whitlock [10]. Assume a plate (w,h) and a cut of length $w' < w$. Cutting the given plate with cut of length w' produces to subplates (w',h) and $(w-w',h)$. Observe that the same two subplates can be obtained by cutting plate (w,h) with a cut of length $w'' = w - w'$. Cut lengths w' and w'' are thus symmetric regarding (w,h). To avoid such symmetries, one can restrict the set of possible cut lengths without missing any patterns by performing only cuts up to the half of the width of a plate. This also holds for the cut performed horizontally. The any-stage dynamic program becomes:

$$F(-,w,h) = \max \left\{ \begin{array}{l} F(0,w,h), \\ F(-,w,h') + F(-,w,q(h-h')), h' \in \mathcal{H}, h' \leqslant h/2 \\ F(-,w',h) + F(-,r(w-w'),h), w' \in \mathcal{W}, w' \leqslant w/2 \end{array} \right\} \quad (1.49)$$

The dynamic programming approach has the advantage to handle well cutting constraints of 2KP. Moreover it is also efficient to solve large instances both for the k-stage and any-stage variants. In Beasley [5], the author compares k-stage and any-stage variants solved by dynamic programming. He solves all instances $gcut1-12$ in seconds and in the same time compares 1-stage, 2-stage, 3-stage and any-stage variants. Due to memory limitation of the computer, author heuristically tackles the remaining instance $gcut13$. The proposed heuristic is to truncate the complete dynamic program and only works on a smaller one. Using a modern computer, it is possible to solve the largest unconstrained instances, $LU1-LU4$ and $LW1-LW4$, in less than five minutes as done by Russo et al. [70].

1.3.3 Exact methods for the bounded 2KP

Dynamic programming is a natural choice when an unbounded 2KP has to be solved. It does not extend easily to the bounded case. However most applications consider only two or three stages. ILP formulations to such special cases may be efficient using modern solvers.

1.3.3.1 ILP based exact methods for two-stage 2KP

Initial works on ILP models for the C-2KP-R-2-f were done by Lodi and Monaci [50]. Authors exploit cutting properties to write their formulations. From the

two-stage restriction, a valid solution to the cutting problem is represented by a set of vertical strips. Each strip is initialized with an item with respect to the restricted cut property. The idea is to find items to set up those vertical strips and then pack extra items vertically without exceeding bin dimension.

Assume a 2KP instance where the number of items is $m = |\mathcal{I}|$. Authors first consider each item $i \in \mathcal{I}$ to be distinct by creating d_i identical items j such that $(w_j, h_j) = (w_i, h_i)$ and $e_j = e_i$. The number of distinct items is $n = \sum_{i \in \mathcal{I}} d_i$. By sorting distinct items in decreasing width, authors assume that n strips may be initialized (the k-th strip is initialized with the k-th distinct item). By defining a binary variable $x_{jk} = 1$ if distinct item j is cut from strip k, 0 otherwise, the following ILP model is obtained:

$$\max \sum_{j=1}^{n} e_j \sum_{k=1}^{j} x_{jk} \tag{1.50}$$

$$\text{s.t.} \sum_{k=1}^{j} x_{jk} \leqslant 1, \quad j \in \{1, \dots, n\} \tag{1.51}$$

$$\sum_{j=k+1}^{n} h_j x_{jk} \leqslant (H - h_k) x_{kk}, \quad k \in \{1, \dots, n-1\} \tag{1.52}$$

$$\sum_{k=1}^{n} w_k x_{kk} \leqslant W \tag{1.53}$$

$$x_{jk} \in \{0, 1\}, \quad k \in \{1, \dots, n\}, \quad j \in \{k, \dots, n\} \tag{1.54}$$

Objective function (1.50) maximizes the profit of cut items. Constraint set (1.51) ensures to cut each distinct item at most one time. Constraints (1.52)-(1.53) are related to bin dimensions. Note that if variable $x_{kk} = 1$, this implies that strip k is used and initialized with item k.

Secondly, the authors write another ILP model however this time decomposition of items is done only by creating a mapping between items and strips. Since cuts are restricted, the number of possible widths for vertical strips is bounded by the number of different item widths. It is then possible to initialize $n = \sum_{i \in \mathcal{I}} d_i$ strips at most (by cutting one item copy in each strip). Assume that items are sorted by decreasing width. The i-th item in $\mathcal{I}$ can be cut from strips of width greater than or equal to w_i. Any item $i = \{1, \dots, m\}$ can be cut from a strip in range $\{1, \dots, \alpha_i\}$ where $\alpha_i = \sum_{s=1}^{i} d_s$ represents the maximum strip index in which it is possible to cut item j. Any strip can be used to cut items in the range $\{\beta_k, \dots, m\}$ with $\beta_k = \min\{r | 1 \leqslant r \leqslant m, \alpha_r \geqslant k\}$ the item that must initialize strip k. Let integer variables x_{ik} be equal to the number of items i cut from strip k if $i \neq \beta_k$ and equal to the number of additional item i cut from strip k otherwise. The second ILP model is:

$$\max \sum_{i=1}^{m} e_i (\sum_{k=1}^{\alpha_i} x_{ik} + \sum_{k=\alpha_{i-1}+1}^{\alpha_i} q_k) \tag{1.55}$$

$$\text{s.t.} \sum_{k=1}^{\alpha_i} x_{ik} + \sum_{k=\alpha_{i-1}+1}^{\alpha_i} q_k \leqslant d_i, \quad i \in \{1, \ldots, m\} \tag{1.56}$$

$$\sum_{i=\beta_k}^{m} h_i x_{ik} \leqslant (H - h_{\beta_k}) q_k, \quad k \in \{1, \ldots, n\} \tag{1.57}$$

$$\sum_{k=1}^{n} w_{\beta_k} q_k \leqslant W \tag{1.58}$$

$$\sum_{s=k}^{\alpha_i} x_{is} \leqslant d_i - (k - \alpha_{i-1}), \quad i \in \{1, \ldots, m\}, \quad k \in \{\alpha_{i-1}, \ldots, \alpha_i\} \tag{1.59}$$

$$x_{ik} \in \{0, \ldots, d_i\}, \quad i \in \{1, \ldots, m\}, \quad k \in \{1, \ldots, \alpha_i\} \tag{1.60}$$

$$q_k \in \{0, 1\}, \quad k \in \{1, \ldots, n\} \tag{1.61}$$

Binary variable $q_k = 1$ if strip k is used, 0 otherwise. As in the previous model, objective function (1.55) and constraint set (1.56)-(1.58) ensure to maximize the profit of cut items without exceeding item availabilities and bin dimension. Redundant constraints (1.59) are used to strengthen the bound on x_{ik} variables.

From these two ILP formulations, authors show how to pass from one to the other. Models are then strengthened with linear inequalities to remove symmetries and several problem variants are considered (rotation, exact case, item lower bounds). Computational experiments outline that both models are able to solve literature instances in a relative small amount of time on average. Tested instances include all constrained literature instances except $CW1-10$,$CU1-11$ and APT set. The approach is also suitable for some unconstrained instances. For unconstrained problems, the authors point out that their models is not the best approach compare to dynamic programming. For constrained 2KP, it is not clear if large instances can be tackled with 2-stage ILP models.

1.3.3.2 ILP based exact methods for three-stage 2KP

Three stage variant is often considered in industrial applications. Indeed increasing the number of cutting stages from two to three allows one to create more patterns, which often results in better use of bin area. Obviously adding an extra cutting stage also results in more complex formulations.

A first attempt on three stage industrial problem was done by Vanderbeck [83] in a column generation context. Since Dantzig-Wolfe decomposition and column generation strengthen ILP formulations for the 2BP, the author solved

the C-2KP-R-3-f in the same way. Using the guillotine cut property, a cutting pattern is decomposed in vertical strips. Each vertical strip is then cut in horizontal sections, each section leads to item cut. The problem can therefore be formulated using Dantzig-Wolfe decomposition. The master problem selects vertical strips and the pricing problem creates them. Sections in vertical strips are obtained using enumeration. Enumeration in this case is possible due to the huge number of industrial requirements that really constrained the problem. A cutting pattern is then obtained by selecting strips. With his approach, the author solves industrial instances but does not mention their size.

A way to solve the C-2KP-NRE-3-f is to write its ILP formulation. Based on the strip decomposition proposed by Lodi and Monaci [50], Puchinger and Raidl [66] extend it to the three stage problem. Formulation uses staged decomposition induced by guillotine cuts. A cutting pattern is decomposed in vertical strips decomposed themselves in horizontal sections containing items of same height. Assume a 2KP instance and as done by Lodi and Monaci [50], each item $i \in \mathcal{I}$ is considered to be distinct leading to the creation of d_i identical items j such that $(w_j, h_j) = (w, h)$ and $e_j = e_i$. The number of distinct items is $n = \sum_{i \in \mathcal{I}} d_i$. By sorting distinct items in decreasing width, n strips and n sections may be initialized. Let binary variable $\alpha_{ji} = 1$ if item i is contained in section j, 0 otherwise and binary variable $\beta_{kj} = 1$ if section j is contained in strip k, 0 otherwise. The ILP model for the C-2KP-R-3-f is:

$$\max \sum_{i=1}^{n} e_i \sum_{j=1}^{i} \alpha_{ji} \tag{1.62}$$

$$\text{s.t.} \sum_{j=1}^{i} \alpha_{ji} \leqslant 1, \quad i \in \{1, \ldots, n\} \tag{1.63}$$

$$\sum_{k=1}^{n} w_k \beta_{kk} \leqslant W \tag{1.64}$$

$$\sum_{j=k}^{n} h_j \beta_{kj} \leqslant H \beta_{kk}, \quad k \in \{1, \ldots, n-1\} \tag{1.65}$$

$$\sum_{k=1}^{j} \beta_{kj} = \alpha_{jj}, \quad j \in \{1, \ldots, n\} \tag{1.66}$$

$$\sum_{i=j}^{n} w_i \alpha_{ji} \leqslant \sum_{k=1}^{j} w_k \beta_{kj}, \quad j \in \{1, \ldots, n\} \tag{1.67}$$

$$\alpha_{ji} = 0, \quad \begin{array}{l} j \in \{1, \ldots, n-1\}, i \in \{j, \ldots, n\}, \\ i > j | h_i \neq h_j \vee w_i + w_j > W \end{array} \tag{1.68}$$

$$\alpha_{ji} \in \{0, 1\}, \quad j \in \{1, \ldots, n\}, \quad i \in \{j, \ldots, n\} \tag{1.69}$$

$$\beta_{kj} \in \{0, 1\}, \quad k \in \{1, \ldots, n\}, \quad j \in \{k, \ldots, n\} \tag{1.70}$$

Objective function (1.62) ensures that profit of cut items is maximized. Item production is bounded by constraints (1.63). Constraint (1.64) limits the width of opened vertical strips to the bin width. Constraint set (1.65) bounds the total height of sections attached to a given strip to not exceed bin height. Constraint set (1.66) forces to cut exactly once an item j in an opened strip k assuming that if section j is opened item j is automatically cut in it. Constraints (1.67) limit the width of opened sections j, represented by the sum of the width of cut items in it, attached to an opened strip k to not exceed strip width. Constraints (1.68) enforce the fact that items cut in the same section must have the same height and that their widths do not exceed bin width due to the restricted cut property. This is mainly used to reduce the number of α variables.

Authors do not report specific results on the behaviour of the 2KP model. Indeed they use it to price columns in a branch-and-price for the 2BP. However tested bin-packing instances up to $B = (300, 300)$ and $|\mathcal{I}| = 100$ are solved by authors. One can assume that three-stage ILP model for the 2KP is able to solve such instances.

1.3.3.3 ILP based exact methods for any-stage 2KP

Models proposed by Lodi and Monaci [50] exploit the structure of the two-stage cutting process. In a more general context authors focus also on solving the C-2KP-NRE-∞-f. A generic way is to formulate this problem as an ILP. Since a guillotine cut divides a plate in two subplates and the number of possible subplates is finite, one can enumerate them as noticed by Furini et al. [30]. Assume a plate $j = (w_j, h_j)$ in which a guillotine cut is performed at position q starting from the bottom left corner of j with vertical v or horizontal h orientation. Possible orientations are grouped in $O = \{v, h\}$. The set $\mathcal{J}$ is used to define the set of all possible plates j with dimension $(w_j, h_j) \in \{1, \dots, W\} \times \{1, \dots, H\}$. The initial plate (W, H) is indexed by 0 in set $\mathcal{J}$. The set $\bar{\mathcal{J}}$ represents the set of plates $j \in \mathcal{J}$ having the same dimension of an item $i \in \mathcal{I}$. (*i.e.* the set of plates such that $(w_j, h_j) = (w_i, h_i), \forall i \in \mathcal{I}$) For a plate $j \in \mathcal{J}$, let $Q(j, v)$ (resp. $Q(j, h)$) be the set of cutting positions $\{1, \dots, w_j - 1\}$ (resp. $\{1, \dots, h_j - 1\}$) which can be performed on plate j with orientation v (resp orientation h). Let integer variables x^o_{qj} be equal to the number of times a plate j is cut at position q with orientation o and with binary coefficient $a^o_{qkj} = 1$ if a plate k is obtained after cutting a plate j at position q with orientation o, 0 otherwise. Using previous notations and with variable y_j the number of times a plate $j \in \bar{J}$ is used, an ILP model for the C-2KP-NRE-∞-f is:

$$\max \sum_{j \in \bar{J}} e_j y_j \tag{1.71}$$

$$\text{s.t.} \sum_{j' \in J} \sum_{o \in O} \sum_{q \in Q(j',o)} a^o_{qj'j} x^o_{qj'} - \sum_{o \in O} \sum_{q \in Q(j,o)} x^o_{qj} - y_j \geqslant 0, \quad j \in \bar{J}, j \neq 0 \tag{1.72}$$

$$\sum_{j' \in J} \sum_{o \in O} \sum_{q \in Q(j',o)} a^o_{qj'j} x^o_{qj'} - \sum_{o \in O} \sum_{q \in Q(j,o)} x^o_{qj} \geqslant 0, \quad j \in J \backslash \bar{J} \tag{1.73}$$

$$\sum_{o \in O} \sum_{q \in Q(0,o)} x^o_{q0} + y_0 \leqslant 1 \tag{1.74}$$

$$y_j \leqslant d_j, \quad j \in \bar{J} \tag{1.75}$$

$$x^o_{qj} \geqslant 0, x^o_{qj} \in \mathbb{N}, \quad j \in J, o \in O, q \in Q(j,o) \tag{1.76}$$

$$y_j \geqslant 0, y_j \in \mathbb{N}, \quad j \in \bar{J} \tag{1.77}$$

Objective function (1.71) maximizes the profit of cut items. Item production is bounded by (1.75). Constraint set (1.72) imposes that the number of plates j that are cut as items does not exceed the number of plates j obtained trough the cut of some other plates. Constraint set (1.73) is the same as the previous one except that a plate cannot be converted into an item. Constraint (1.74) ensures that the initial plate is not used or cut at most one time. Constraint set (1.75) bounds the production of item.

This ILP formulation has a pseudo-polynomial number of variables x depending on possible ways to cut a plate. A straightforward way to reduce this number is to apply pattern normalization. However this reduction procedure cannot reduce the number of variables sufficiently. To solve their cutting problem, the authors propose to first use a reduced formulation by using the restricted cut property. A C-2KP-RE-∞-f is solved instead of a C-2KP-NRE-∞-f. This reduces the number of positions where the plate can be cut, and the number of variables. The drawback is that suboptimal solutions for where the plate can be cut.he initial problem can be obtained. Since authors want to solve the C-2KP-NRE-∞-f but are limited by the number of variables, they propose to incrementally solve it using variable pricing. The idea is to incrementally add x variables with a positive reduced profit when solving the linear relaxation of (1.71)-(1.77).

Model (1.71)-(1.77) is first initialized with x^o_{qj} variables related to restricted cuts. Then by solving the linear relaxation of (1.71)-(1.77), dual variables π_j are obtained for constraints (1.72)-(1.73) and one computes for all x^o_{qj} variables its reduced profit:

$$\bar{p}(x^o_{qj}) = \pi_{j1} + \pi_{j2} - \pi_j \tag{1.78}$$

The computed reduced profit $\bar{p}(x^o_{qj})$ corresponds to the change in the objective function for a unitary increase in the value of the corresponding variable. Variables with positive reduced profit are then added to the restricted model (1.71)-(1.77). Process is repeated until no more variable with positive reduced

profit can be added to the model. Finally the last pricing is done for variables x^o_{qj} such that:

$$\left\{x^o_{qj}, j \in J, o \in O, q \in Q(j,o) | \lfloor \bar{p}(x^o_{qj}) + LP \rfloor > LB \right\} \tag{1.79}$$

LP is the optimal solution of the linear relaxation of (1.71)-(1.77) after variable pricing procedure and LB a problem lower bound obtained with a heuristic. The reduced model is then solved with a commercial solver.

Computational experiments show that the variable pricing approach performs well and reduces the model size. Authors test their approach on all constrained literature instances except APT set. They also test it on unconstrained set *gcut* by setting a demand of one to each item. They are able to solve $gcut10 - 12$, *i.e.* instances with a $B = (1000, 1000)$ and $\sum_{i \in \mathcal{I}} d_i = 50$. However they fail to outperform existing state-of-the-art approach of Dolatabadi et al. [23] due to a too large model size and computational time limit.

1.3.3.4 Recursive exact algorithm for any-stage 2KP

Best performances on the C-2KP-NRE-∞-f are obtained by Dolatabadi et al. [23]. Their algorithm is based on a dynamic program combined with an implicit enumeration of patterns. Implicit pattern enumeration uses the definition of a feasible packing and maximal packing for a plate. Given a 2KP instance, the number of different items in the instance is equal to $m = |\mathcal{I}|$. By abuse of notations, items are assumed to be indexed by position $j = \{1, \ldots, m\}$. A feasible packing f is represented by a non negative vector $[f_1, f_2, \ldots, f_m]$. Each value $f_j \leqslant d_j$ is the number of cut items of index $j \in \{1, \ldots, m\}$ in packing f. The profit of a feasible packing f is the sum of profits of cut items in it:

$$p(f) = \sum_{j=1}^{m} e_j f_j \tag{1.80}$$

A feasible packing is said to be maximal if there are no other feasible packing f' such that $p(f') \geqslant p(f)$. For two feasible packings f and f', a new packing $\tilde{f} = f + f'$ is defined as follows:

$$\tilde{f_j} = \min\left\{f_j + f'_j, d_j\right\}, j = \{1, \ldots, m\} \tag{1.81}$$

Using packing definition, one can define the set $F(w, h, z)$ containing all feasible packings of profit larger than or equal to z for a plate (w, h). Let the pairwise sum $F^1 \oplus F^2$ of two feasible packings be:

$$F^1 \oplus F^2 = \left\{ f + f', \forall f \in F^1, \forall f' \in F^2 \right\} \tag{1.82}$$

Since cuts are guillotine, the algorithm proposed by the authors recursively computes sets $F(w, h, z)$ for all $w \in \mathcal{W}$ and $h \in \mathcal{H}$. Indeed feasible packings contained in $F(w, h, z), w > 0, h > 0$ are obtained by combining feasible packings from $F(w', h, z)$ and $F(w - w', h, z)$ with $w > w' > 0$ or from $F(w, h', z)$ and $F(w, h - h', z)$ with $h > h' > 0$. To reduce the dynamic program size, sets $\mathcal{W}$ and $\mathcal{H}$ are normalized (see Christofides and Whitlock [10]).

A drawback of the algorithm is its long running time since all feasible packings are enumerated for each $F(w, h, z)$. A straightforward improvement is to compute an upper bound $U(f)$ on the maximum profit that can be obtained in the residual area of the plate for a feasible packing $f \in F(w, h, z)$. Such bound is obtained by solving a 1KP instance where each item i is available $d_i - f_i$ times with a profit e_i and a weight $w_i \times h_i$, bin capacity is set to $W \times H - w \times h$. A feasible packing f is then discarded if inequality $p(f) + U(f) < z$ holds, where z represents the value of a known feasible solution. In other words, packing f cannot be used to obtain a solution of value larger than z.

Using the pattern enumeration algorithm, the whole method proceeds by trial-and-error on the objective value estimate. Assuming a given threshold value th to be a valid lower bound, feasible cutting patterns are enumerated by the recursive method, using value th to fathom patterns. If a feasible packing is found using this threshold value, the associated cutting pattern is optimal. Otherwise the threshold value is too high and all optimal packings have been discarded by fathoming. One needs to solve the problem again after decreasing the threshold value. Authors start their algorithm using an initial threshold value th equal to a problem upper bound UB. If algorithm fails to find an optimal feasible packing, threshold value is decreased by $(UB - LB)/10$ with LB a heuristic problem lower bound.

Computational experiments outperform all existing approaches for the C-2KP-NRE-∞-f. The algorithm is able to provide the optimal solution for almost all instances $APT30-39$ and $APT40-49$. They also test their algorithm on W and *okp* set and transformed *gcut* instances. The maximum running time is set to one hour but most of instances are solved in less than 50 seconds. The main drawback of the algorithm is its dependence to initial lower and upper bounds.

1.3.3.5 Branch-and-bound approaches

A way to solve 2KP is to apply a branch-and-bound algorithm using the guillotine property. Since a guillotine cut divides a plate in two subplates, an intuitive branching decision is to decide on the position to perform such cut.

By iteratively branching on vertical and horizontal positions, the optimal cutting pattern can be obtained. This branching is done in a top-down way. Initial branching decision is done on plate (W, H) and then on produced subplates. However the tree size is exponential and one needs to select a good branching strategy to avoid useless exploration of the tree. This first attempt was made by Christofides and Whitlock [10] and then improved by Christofides and Hadjiconstantinou [9]. Later Morabito and Arenales [58] solved the C-2KP-NR-k-f using an AND/OR graph representation. Each vertex in this graph represents a possible plate of given dimension. Each arc in this type of graph is called an AND arc. Formally it is an oriented connection between one vertex and a set of vertices. Applied to cutting problem, such an arc represents a cutting decision. An AND arc gives indication on how to cut a given plate into two subplates. The OR part of this graph, close to the logical OR operator, is used to limit the number of AND arcs to select for a given vertex in the AND/OR graph. Among the set of possible AND arcs for a given vertex, only one can be selected. Concretely this forces to perform only one guillotine cut on a given plate. Consequently among the set of possible cuts which can be performed on a given plate, only one can be retained. A representation of AND/OR graph is given in Figure 1.12. The problem representation with AND/OR graph gives rise to a classical top-down branch-and-bound.

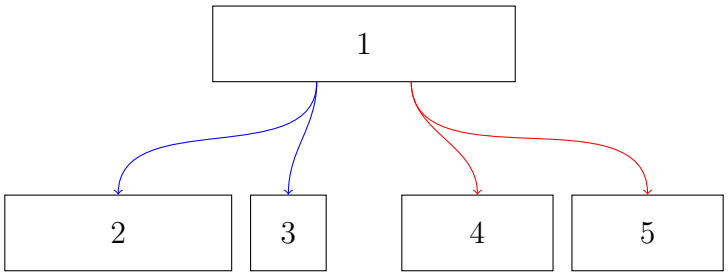

Figure 1.12: Representation of two AND/OR arcs. The first one is between plate 1 and subplates 2 and 3. The second is between plate 1 and subplate 4 and 5.

Another way to perform branching is to do it in a bottom-up fashion. The idea is to maintain a list of valid patterns, initialized with item set $|\mathcal{I}|$, and to try to create new patterns by combining two previously created patterns. The creation of a new pattern is performed in two steps. First the two patterns to combine are checked to ensure that items are not overproduced. Then one has to ensure that the two patterns put side by side or on top of the other do not exceed the bin dimension. The bottom-up branch-and-bound was first proposed by Hifi and Roucairol [42] on C-2KP-NR-2-f and C-2KP-NRE-2-f and then used by G et al. [31] for U-2KP-NRE-2-f.

Computational experiments conducted by previous authors show that branch-and-bound is an efficient way to obtain optimal solutions. Nevertheless, con-

sidered instances in Hifi and Roucairol [42] and Morabito and Arenales [58] are $OF1-2$, $Hchl$, CHL, $A1-5$. For these instances, the bin size is around 200×200. Their approaches sometimes fail to find the optimal solution within a given time limit. Some instances are not solved to optimality for the 2-stage problem in less than two hours in Hifi and Roucairol [42]. For the unbounded variant considered by G et al. [31] results are better. Authors provide optimal solutions for $UW1-11$ and $UU1-11$ instances in few seconds.

1.3.3.6 Constraint programming

In addition of ILP techniques, the constraint programming is also a way to solve 2KP. In Clautiaux et al. [12], authors outline a paradigm based on graph representation of a cutting pattern and constraint programming. They focus on deciding if a given item set $\mathcal{I}$ can be cut from a given bin B using only guillotine cuts. This decision problem is important since it is used when one wants to solve 2SP and 2KP.

The constraint programming approach relies on defining a set of variables, a set of possible values for each variable and a set of constraints between variables. If one finds a set of values for each variable ensuring that constraints are satisfied, the associated problem is solved. A strength of the constraint programming is to use constraint propagation. The idea is to do certain deductions from the domain of variables leading to deduce new constraints or detect inconsistencies. Valid deductions allow to reduce the required computational effort. In order to do a strong and efficient constraint propagation, the authors defined a new graph representation of a guillotine pattern.

Computational experiments show that constraint programming and guillotine graph outperform existing state-of-the-art methods for the 2SP. However used benchmarks only considered small item sets ($|\mathcal{I}| = 23$ at most) and small bins ($H = 175$ at most).

1.3.4 Heuristic approaches

Exact methods for the bounded 2KP can be slow. Moreover, those methods often need to be initialized with a good quality incumbent solution. To obtain such incumbent heuristics are used. Heuristics for the 2KP are often divided in two categories: top-down and bottom-up. In top-down heuristics, an initial bin is cut iteratively to obtain items. In bottom-up heuristics, initial items are combined together to create partial valid patterns, which are themselves combined to obtain a feasible pattern for the initial bin. This section is decomposed in two parts, one for each approach.

1.3.4.1 Bottom-up heuristics

The first bottom-up heuristic has been proposed by Wang [86] for the C-2KP-NR-∞-f. In this work, the author notices that when a guillotine cutting pattern is built, items to cut are just put next to each other. Therefore blocks of items can be merged recursively to create a complete cutting pattern for a given bin. An example of block building is represented in Figure 1.13

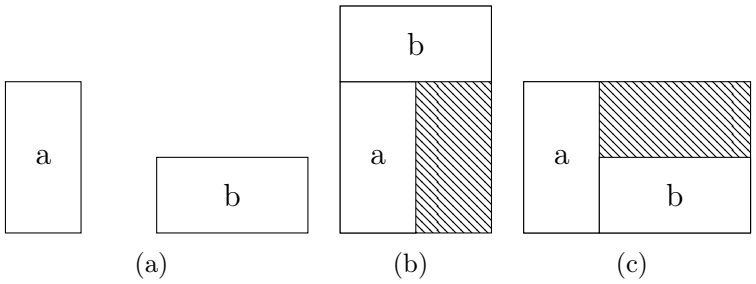

Figure 1.13: Partial plates (a) can be put on top of each other or next to each other to obtain plates (c) and (d)

Some partial patterns can be easily discarded because of item overproduction or because they are larger than the bin dimension. Nevertheless the number of possible partial patterns which can be created or merged remains exponential. Wang [86] proposes to set an acceptance parameter on the waste contained in a partial pattern to avoid the exponential growth This forces the building procedure to only consider partial patterns with a waste value lower than a given parameter. Later works of Oliveira and Ferreira [62] improve the condition on how to discard pattern.

The bottom-up approach for any-stage problem can be modified to handle 3-stage problem, as outlined by Cui et al. [16]. Since an item has to be cut in at most three cuts, authors observe that a cutting pattern decomposes itself in vertical strips, which are themselves decomposed in horizontal segments. Segments are then cut to produce items. Authors decide to solve this problem in a bottom-up way by solving three dynamic programs, one for each cutting stage. Initial strips are produced using an exact dynamic program, then segments and items are created by heuristic dynamic programs. They are used by authors to handle item upper bounds and also to work on a smaller state space size.

The bottom-up approach for three-stage problem is a suitable approach. Indeed authors in Cui et al. [16] solve APT instances in seconds. They also test their approach on a custom set with a bin size $B = (3000, 1500)$ and a number of different items $|\mathcal{I}| = 150$.

1.3.4.2 Top-down heuristics

Top-down heuristics aim to divide a given plate in two subplates recursively. The standard way to solve a cutting problem with a top-down approach is to first design a heuristic to produce a feasible pattern and then to include it in a meta-heuristic.

A first application is due to Alvarez-Valdés et al. [2] and uses a Greedy Randomized Adaptive Search Procedure (GRASP), a Tabu Search (TS) and a Path-Relinking (PR) for the U-2KP-NRE-∞-f and C-2KP-NRE-∞-f. The initial constructive heuristic cuts iteratively items in $\mathcal{I}$. Given a plate, the selected item to cut is the one which has the best upper bound. When a given item is cut in a given plate, at most two subplates are produced. An upper bound on the value of a solution that can be obtained by this cut is computed by summing up the item value and the upper bound of each obtained subplates. To compute such upper bound, the 1KP relaxation of the 2KP is used. A direct GRASP to obtain different cutting patterns consists in randomly selecting an item to cut depending on some parameters. Starting from an initial cutting pattern p the authors defined a Tabu Search procedure. The idea is to start from pattern p, select some subpatterns and remove them from p. This creates holes in pattern p. They are then filled by using the GRASP procedure. Since one wants to avoid to recreate the same subpatterns that the one which have been removed, they are stored in a tabu list. When the GRASP is used to fill the holes, only partial patterns not stored in the tabu list are considered to be valid. This diversifies produced cutting patterns. Better solutions are also obtained by mean of a PR procedure in which an initial pattern p is transformed into a second one p'. The initial cutting pattern p is obtained by a set of guillotine cut. The PR procedure creates an initial empty pattern and then applies on it a subset of guillotine cuts used to obtain p. This gives a partial pattern which is then filled using the GRASP procedure to obtain a new complete cutting pattern p'.

Computational experiments of Alvarez-Valdés et al. [2] show that their approaches are suitable for APT sets. Nevertheless for large instances with $(W, H) = (1000, 1000)$, average computation times for TS are high. Thus, the initial constructive heuristic or only the GRASP are preferred to obtain solution in a reasonable amount of time.

1.3.4.3 Primal-dual heuristics

Constructive heuristics start from an empty solution and iteratively construct a valid one. Some solutions can be difficult to find using pure primal heuristics like constructive ones or meta-heuristics. To avoid this drawback and to increase the chance to obtain good quality solutions and in some cases prove the solution optimality, a suitable approach is to use primal-dual heuristics. The aim of this type of heuristics is to combine constructive process as in classical

heuristics but also exploit dual information to drive constructive choices in a promising direction.

The initial contribution for the 2KP problem was first done by Christofides and Whitlock [10] for C-2KP-RE-∞-f. In their first attempt to solve the constrained 2KP, authors outline that the standard dynamic programming approach is not suited when the number of items is large. The authors state that it is possible to write a dynamic program for the bounded 2KP based on the unbounded one. Each state of the unbounded dynamic program is extended by one extra dimension for each possible item to cut. An initial state represented by two components (*i.e.* the plate dimension (w, h)) is now represented by $2 + |\mathcal{I}|$ components (*i.e.* the plate dimension (w, h) and one extra dimension for each item). Clearly the bounded dynamic program for the 2KP is intractable due to its size. To avoid the combinatorial explosion, an approach called Decremental State Space Relaxation (DSSR) is used. The starting point is a weak relaxation of an initial problem to solve. Since the problem is relaxed it is easier to solve compared to the original one. Nevertheless since it is a weak relaxation, its optimal solution may be of poor quality. The idea of the DSSR is to iteratively strengthen this weak relaxation by including some relaxed constraint. At each iteration, the new relaxed problem becomes closer to the initial one, but also harder to solve. The main motivation of the DSSR is to solve optimally relaxed problems and then prove that the optimal relaxed solution is also optimal for the initial problem. In the case discussed in Christofides and Hadjiconstantinou [9], the authors relax the initial bounded 2KP by first solving the unbounded 2KP with dynamic programming. They then extend the unbounded dynamic program by adding an extra dimension to it. This gives a better relaxation of the bounded 2KP. The purpose of this extra dimension is to bound the total item production by a value $D = \sum_{i \in \mathcal{I}} d_i q_i$ using a non negative weight q_i for each item. Initial weights are set to zero and are then adjusted using a subgradient method. At each subgradient iteration, the optimal relaxed solution from the extended dynamic program is checked to ensure its validity regarding item demand constraints. This may improve a best known incumbent. Thus, at the end of the subgradient, the best found primal solution as well as the best dual solution are stored among the ones found during subgradient iterations. Authors use this information to initialize their top-down branch-and-bound to obtain optimal solution. Authors outline that during subgradient process, it is possible to prove optimality of the primal solution if the primal solution value matches the dual bound value.

The primal-dual heuristic using subgradient for the 2KP is able to solve problem instances. Nevertheless it suffers from the lack of efficient method for subgradient weight adjustment. For example at a given subgradient iteration, the weight adjustment may lead to a solution of the dynamic program already found some iterations ago. To avoid this effect and to obtain stronger dual bounds, Velasco and Uchoa [85] improve the previous primal-dual heuristic by

forbidding previous dual solutions to appear again. The subgradient weight adjustment is substituted with solving an ILP model instead. The purpose of this ILP is to find new weights for all items which exclude previous found infeasible solutions and to minimize the value D. In a second step the authors strengthen their own approach by adding two weight vectors and consequently two extra dimensions to the dynamic program. They therefore extend the ILP model used to find weight vectors. Authors also develop a heuristic which looks for a feasible primal solution based on the relaxed dynamic program. They bound their algorithm by a limited number of iterations and a threshold value on D. Computational experiments on C-2KP-RE-∞-f outlined that the method provides near optimal solutions.

The idea of introducing a feasibility heuristic when finding iteratively weight vector has also been used by Morabito and Pureza [59]. The retained approach of authors is the one of Christofides and Hadjiconstantinou [9] but for each weight vector a feasibility heuristic starts to find a better primal solution. Contrary to Velasco and Uchoa [85], the feasibility heuristic aims to repair the infeasible solution obtained at a given subgradient iteration.

Results mentioned by Velasco and Uchoa [85] are mitigated. Indeed for instances $CU1-11$,$CW1-11$,$OF1-2$ and $CHW1-3$, their method clearly outperforms the best known one of Dolatabadi et al. [23]. However, on average it cannot solve large instances ($ATP30-39$ and $APT40-49$) in a short amount of time.

1.4 Two-dimensional bin-packing problem

There are different ways to solve 2BP such as exact and heuristic methods. This section describes the main ones. An overview of 2BP instances is first done. A description of exact methods to solve the 2BP is given next. This is then followed by a description of heuristic strategies.

1.4.1 Problem instances

The major dataset to computationally compare methods for solving 2BP has been proposed for the classical non-guillotine 2BP. But these instances are also used for the guillotine variant of the problem by adding the corresponding constraint. Instances are divided in ten classes. Let $\chi \sim U[\alpha, \beta]$ denote that value χ is uniformly generated in interval $[\alpha, \beta]$. Six instance classes were proposed by Berkey and Wang [7] and are built as follows:

Class 1: $W = H = 10$, $w_i, h_i \sim U[1, 10]$

Class 2: $W = H = 30$, $w_i, h_i \sim U[1, 10]$

Class 3: $W = H = 40$, $w_i, h_i \sim U[1, 35]$

Class 4: $W = H = 100$, $w_i, h_i \sim U[1, 35]$

Class 5: $W = H = 100$, $w_i, h_i \sim U[1, 100]$

Class 6: $W = H = 300$, $w_i, h_i \sim U[1, 100]$

The four remaining classes with $W = H = 100$ were created by Martello and Vigo [55]. Authors divided items in types:

Type 1: $w_i \sim U[2/3W, W]$, $h_i \sim U[1, 1/2H]$

Type 2: $w_i \sim U[1, 1/2W]$, $h_i \sim U[2/3H, H]$

Type 3: $w_i \sim U[1/2W, W]$, $h_i \sim U[1/2H, H]$

Type 4: $w_i \sim U[1, 1/2W]$, $h_i \sim U[1, 1/2H]$

The four classes are:

Class 7: Type 1 with probability 70%, Type 2, 3, 4 with probability 10% each

Class 8: Type 2 with probability 70%, Type 1, 3, 4 with probability 10% each

Class 9: Type 3 with probability 70%, Type 1, 2, 4 with probability 10% each

Class 10: Type 4 with probability 70%, Type 1, 2, 3 with probability 10% each

In these instances, the total number of item copies ($\sum_{i \in \mathcal{I}} d_i$) in each instance belongs to $\{20, 40, 60, 80, 100\}$. Remark that for Class 2,4,6,7,8,10, the size of items is small compare to the bin dimension. 10 instances for each class are generated. Thus, the dataset contains 500 instances. They are available at `http://or.dei.unibo.it/library/two-dimensional-bin-packing-problem`.

1.4.2 Pseudo-polynomial size ILP formulations

A possible way to solve 2BP is to extend 2KP formulations. The modified objective function is to minimize the number of bins to use instead of maximizing the profit of one bin. In Lodi and Monaci [50], the authors introduce two-stage 2KP formulations. In later works described in Lodi et al. [49], they rewrite the objective function of their initial 2KP models to match the objective function of 2BP. The rewriting of the objective function from a 2KP model to a 2BP model is also outlined in Furini et al. [30]. The considered 2BP is any-stage in their case.

There also exist pseudo-polynomial size ILP formulations not based on existing ones for the 2KP. In Silva et al. [75], the authors enumerate the whole set of ways to cut a given plate into subplates. Starting from this set, the authors write ILP formulations for the two-stage and three-stage 2BP. Each variable in their models corresponds to a way to cut a plate into subplates. In Puchinger and Raidl [66], the proposed ILP formulation is similar to the one described in Lodi et al. [49]. The main difference is that considered problem is three-stage instead of two-stage leading to more variables in the proposed model.

An original problem formulation is proposed by Macedo et al. [52]. Starting from the flow formulation of Valério de Carvalho [79] for the 1BP, they extend it to the two-stage problem. An oriented acyclic graph is created to represent the sequence of different vertical strips to cut from a bin. Then for each possible vertical strip of given width, extra oriented acyclic graphs are created to represent the different items to cut. Using variables to link graphs between them, authors write a flow formulation to solve the 2BP.

Experiments done in Lodi et al. [49] focus on literature instances. Their model obtains good results. Nevertheless for instances in which the size of items is small compared to the bin size, they do not find the optimal solution in five minutes. In Puchinger and Raidl [66], the authors compared their models for the three-stage variant with the two-stage ones in Lodi et al. [49] on literature instances. Results are nice since their three-stage model solves to optimality as many instances as the two-stage model in around one minute on average. Puchinger and Raidl [66] then compared their three-stage models for the restricted and unrestricted cases. The unrestricted model has worse results than the restricted one. This is mainly due to the number of variables which is larger when cuts are unrestricted. In Silva et al. [75], the computational experiments have not been done on 2BP literature instances. The authors use modified 2KP literature instances instead. Nevertheless, they compare their two and three-stage models based on plate enumeration with the ones of Lodi et al. [49] and Puchinger and Raidl [66]. Results are competitive for both methods without clear dominance of one models compared to the others. In the ILP formulation based on graphs introduced in Macedo et al. [52], the authors conduct experiments on instances taken from the furniture industry. They divide their instance in two sets A and B. For the first one, on average, the bin size is $(1875, 2600)$, the number of different items $|\mathcal{I}| = 29$ and the total number of items to cut $\sum_{i\in\mathcal{I}} d_i = 199$. The dataset B has larger bin size $B = (1935, 4030)$ and total number of items to cut $\sum_{i\in\mathcal{I}} d_i = 1759$ but smaller different items $|\mathcal{I}| = 9$ on average. For both datasets, a comparison is done with the two-stage model from Lodi et al. [49]. The flow formulation is clearly the best one to solve two-stage 2BP both in terms of number of solved instances and computation time.

1.4.3 Heuristic approaches

When the 2BP has be to be solved quickly for example to initialize an exact method, one needs to obtain good heuristic solutions. Most heuristics in the literature are greedy and are classified in two types: one-phase heuristics which directly pack items into bins, and two-phase heuristics which start by packing items in strips and then assign strips to bins. First, standard one-phase and two-phase 2BP heuristic are described. Then some meta-heuristics and heuristics based on ILP are detailed. A more complete review of most popular 2BP heuristics can be found in Lodi et al. [48].

1.4.3.1 One-phase heuristics

One-phase heuristics aim to directly pack items into bins. This is typically achieved by selecting an empty plate in which the item fits. Recall that a guillotine cut always divide a given plate in at most two plates of smaller dimensions or subplates. The term subplate used hereinafter represents both an initial plate of size equal to a given bin size and the obtained subplates after performing a guillotine cut on some plate. Difference between heuristics is therefore located in the way a subplate is selected to pack an item. Most heuristics described here were initially developed by Berkey and Wang [7].

A straightforward way to obtain quickly a feasible packing is the Next-Fit heuristic: among a set of empty subplates, a given item is packed in the next subplate in which it fits. Subplates in which the item does not fit are discarded and never used for inspection of other items. If the item does not fit in any subplate, a new bin is opened. The initial set of subplates is initialized with a plate related to a bin. This heuristic is very fast in practice but is very dependent on the initial item order. To correct this behaviour, the First-Fit heuristic is often preferred: among a set of empty subplates, a given item is packed in the first subplate in which it fits. Subplates in which the item does not fit are kept and can be used for inspection of other items. If the item does not fit in any subplate, a new bin is opened. This second heuristic is less myopic since subplates are kept for further inspection. A last strategy to pack items is to use a Best-Fit strategy: among a set of empty subplates, a given item is packed in the subplate in which it fits the best. If the item does not fit in any subplate, a new bin is opened. The main difference with the First-Fit heuristic is the choice of the subplate to use. The notion of best subplate may depend on the residual area obtained if the item is cut or if the item area is not so small compare to the subplate area for example. Finally a Bottom-Left heuristic can also be used to obtain a feasible packing. As mentioned by its name, its purpose is to pack an item among subplates in which it fits such that the item is packed in the lowest and left position. Note that heuristics described here can easily been adapted to solve 2KP variants.

Experiments from Berkey and Wang [7] have been done on Class 1-6 from

the 2BP literature instance set. They are no strict comparison between each method in term of computation time. It is not very critical since described one-phase heuristics run in at most $\mathcal{O}(n^2), n = \sum_{i \in \mathcal{I}} d_i$. Nevertheless best solution quality are obtained by First-Fit and Best-Fit heuristics.

1.4.3.2 Two-phase heuristics

Unlike one-phase heuristics, two-phase heuristics aim to first create good strips in which items are packed and then assign them to bins in order to create a feasible packing for a 2BP instance.

A standard way to create strips is to transform the initial 2BP instance to a 2SP instance. Strips are then obtained by using a one-phase heuristic on the 2SP instance. When all items are packed, the obtained solution is decomposed in a set of vertical strips. They are then assigned to bins by applying one-phase heuristic for a 1BP instance. Strips are characterized by their width and assigned to bin of capacity W. There are therefore two possible parameters to change heuristic behaviour. The first one is related to the 2BP one-phase heuristic used to create strips. The second is related to the way to assign strips to bins by mean of a 1BP one-phase heuristic. Common two-phase heuristics are Hybrid First-Fit and Hybrid Best-Fit heuristics proposed by Berkey and Wang [7]. Strips are first created and then assigned to bins using a First-Fit (resp. Best-Fit) strategy.

Another way to create strips is to solve knapsack problems instead of using heuristics as proposed by Lodi et al. [47]. In the proposed heuristics, the authors first initialized a strip by packing an item i in it. This gives a strip of width w_i and height H. Then they solve the packing problem related to the unused space in this strip as a 1KP_b with a bin of capacity $H - h_i$ and item set $\bar{\mathcal{I}} = \{i' \in \mathcal{I} - \{i\} | h_{i'} \leqslant H - h_i, w_{i'} \leqslant w_i\}$. The strip filling is related to solve a NP-hard optimization problem which can be long in practice. Authors defined a time limit for that case but their experiments outlined that this was not required since solving the 1KP_b subroutine turned to be fast in practice.

Experiments done in Lodi et al. [47] compare the proposed heuristic using knapsack subroutine with the Hybrid First-Fit and Hybrid Best-Fit from Berkey and Wang [7]. To benchmark the heuristics, all 2BP literature instance are used. There are no clear dominance of one heuristic compare to the others but computation time for all of them is around some seconds.

1.4.3.3 Meta-heuristics

The drawback of pure heuristics is that the output solution quality may be far from the optimal solution. To avoid this behaviour, meta-heuristics are used. They embed fast pure heuristics to explore more solutions. If more solutions are explored, the probability to find an improving solution increases.

Purpose of a meta-heuristic is to explore the solution space in a clever manner. An efficient procedure to do it proposed by Polyakovsky and M'Hallah [65]. Authors first define a bottom-left subroutine to solve 2KP. Then by using agent-based methods from simulation theory, an incumbent solution is improved. Agent-based approaches aim to simulate a complex behaviour of a given system starting from an initial simple behaviour from all agents in the system. Authors implement this model by defining two types of agents: an agent-initiator which wants to attract as many as possible other agents to it and an individual-agent which is either free or attached to an agent-initiator. Each individual-agent in the system corresponds to an item to pack. The system is initialized with $\sum_{i \in \mathcal{I}} d_i$ individual-agents and spontaneously some agents decide to become agent-initiator. Those agents try then to attract individual-agents to them in order to create a feasible 2KP solution of best value. Since the system is dynamic, a group composed of an agent-initiator and its following individual-agents can be broken. In that case all agents in the group become free individual-agents. A group breaking only occurs when the quality of the packing using agents in this group is weak. The system is said to be stable when there are no more free individual-agents. In that case using all agent-initiators, a 2BP solution is obtained. Authors run their system many times to diversify the quality of the solution by means of group breaking. Computational experiments shown good results and small computational times are obtained. Authors also outlined that their method is less sensitive to problem parameters.

Another type of meta-heuristic is proposed by Cui et al. [15] based on creation and selection of different 2BP solutions. The idea is to create iteratively z different 2BP solutions and record the best one. A valid solution is created by finding one cutting pattern for each bin. A cutting pattern for a bin is obtained using unbounded dynamic programming to create vertical strips and truncated bounded dynamic programming for each strip. When a complete solution for the 2BP is built, profit e_i of each item is changed to generate different solution at the next iteration. The best found solution is then returned after creation of z solutions.

Some meta-heuristics also rely on partial enumeration to obtain a feasible solution. In Lodi et al. [51] authors develop a method based on an enumeration tree. At each level of the tree a feasible packing for a bin is created. When a leaf is reached a complete 2BP solution is obtained. To obtain a feasible packing for one bin authors implement the Bottom-Left heuristic. To avoid same patterns creation by the Bottom-Left heuristic different packing strategies are used. Their aim is to influence the order in which items will be packed by the heuristic and how the cut will be performed. Since the size of the enumeration tree can be huge, authors also define rules to removed duplicate patterns and heuristic pruning rules.

Computational experiments of all mentioned authors in this section outline

that all approaches are efficient to solve all 2BP literature instances both in term of solution quality and computation time.

1.4.3.4 ILP based heuristics

The Dantzig-Wolfe decomposition applied on 2BP gives a strong lower bound on the optimal integer solution value. A drawback is that after column generation convergence, the obtained solution may be fractional. Since branching to find the optimum one can be long, one wants to use heuristics instead. ILP based heuristics use information from a fractional solution to build an integer one.

A classical way to obtain an integer solution from a fractional solution is achieved by simply rounding up or down some fractional variables to integer values. This approach was tested by Alvarez-Valdes et al. [3] for the 2BP with any-stage patterns. The authors first write the Dantzig-Wolfe reformulation of the 2BP and solve it by column generation. Since the pricing problem to deal with is NP-hard, the authors create heuristic patterns instead. They are obtained using heuristics previously detailed in Alvarez-Valdés et al. [2]. After column generation convergence, the fractional master solution is analysed and all fractional variables are rounded-down. This leads to have some uncut items. To obtain a valid solution, the authors start a heuristic on the fixed solution and cut remaining items.

The round-down strategy is also used in Cintra et al. [11] to solve the 2BP with two-stage and four-stage cutting patterns. The difference with the approach described in Alvarez-Valdes et al. [3] is that authors relax the pricing problem. Usually it is a bounded 2KP but authors solve its unbounded variant. This may produce non-proper columns, meaning that the found pattern does not satisfy bounds on the item production. The main motivation is that using dynamic programming to solve the unbounded 2KP instead of solving to optimality the bounded 2KP is much faster. However since non-proper columns are inserted in the master problem, they cannot be part of any feasible solution for the 2BP. To avoid that, they round-down each column in the optimal fractional solution of the master problem after convergence. The described algorithm is globally similar to the diving heuristic with a main difference related on how to handle the subproblem. The authors notice that when all rounded-down fractional columns are equal to zero, no more columns can be fixed. A modified Hybrid First-Fit heuristic starts in that case to complete the partial fixed master solution.

A direct application of the diving heuristic for the 2BP with two-stage patterns is described in Furini et al. [29]. The retained way to deal with the two-stage pricing problem is to first solve it heuristically and then use the ILP formulation from Lodi et al. [48]. After column generation convergence, the diving heuristic rounds up the fractional variable with the largest value in

the master problem. In the proposed diving heuristic, the master problem is only solved to optimality at the root node. The pricing problem is only solved heuristically in the diving heuristic. Authors also described a way to solve the problem by performing diving heuristic with heuristic columns and then solved the 2BP as an MILP with all columns found during the diving.

Computational experiments in Alvarez-Valdes et al. [3] were done on a randomly set of instances. The largest proposed instances is related to a bin of size $(2000, 1000)$, $|\mathcal{I}| = 50$. An item has dimension at most equal to $(1000, 500)$ and a random demand in range $[100, 200]$. On the proposed dataset, authors outline that the time required to find a solution is highly dependant on the heuristic used to price the columns. Simple constructive heuristics take a small computation time but the quality of the found solution is bad. Using a GRASP heuristic to price columns is the best trade-off found by authors between solution quality and computation time.

The dataset used in Cintra et al. [11] is also different from the standard one of the literature. It is inspired from instances $gcut1 - 12$ described in Beasley [5]. They are modified by setting a random demand in range $[1, 100]$ to each item. The authors from Cintra et al. [11] outline that using non-proper columns to solve the master problem to optimality decreases only slightly its value in comparison to the case when only proper columns are used. The approach is suitable for large instances for 2BP with both two-stage and four-stage cutting patterns with rotation. Authors in Furini et al. [29] benchmark their diving heuristic with the one from Cintra et al. [11] for the two-stage 2BP with rotation. They outline that the diving heuristic with heuristic columns obtained worse results than the approach described in Cintra et al. [11]. Nevertheless using diving heuristic followed by MILP solving gives the best results.

1.5 Variants of two-dimensional cutting problems

Although the 2BP is a well-studied problem in the literature, most papers consider the objective function that is far from industrial considerations. Bins are often pieces of raw material which can be expensive to produce and/or contain defects related to the production process.

In its standard application, the 2BP minimizes the number of bins required to cut an item set. Implicitly this reduces the waste among used bins. For low quality raw material, wastes are often discarded and/or recycled. For high quality material, it is interesting to keep them for another usage such as cutting future items. Nevertheless at an industrial scale, it may be time consuming to keep track of all wastes produced during the cutting process. Since most industrial applications consider guillotine cuts, the widespread waste consideration is the one related to wastes obtained after first stage cuts. In other

words and if cuts are assumed to be done from left to right, cutting patterns have to ensure that the unused right bin part or leftover is maximal. This has the advantage to require a small effort to handle. This variant of the 2BP is called the 2BP with leftover ($2BP_l$). Remark that leftovers can be considered for all used bins or only the last used one in a $2BP_l$ solution. If not mentioned, the notation $2BP_l$ refers to the leftover maximization of only the last cut bin.

To ensure a good quality of service, it is important to cut defect free products. Indeed due to production process, some bins may contain defects. To guarantee that customers are satisfied, the creation of cutting patterns has to handle the set of defects which may occur on bins. In other words, the cutting process has to guarantee that all cut items are defect free and that defects are located in waste bin parts. This variant of the 2BP is called the 2BP with defects ($2BP_d$).

Having the leftover and defect consideration in mind, a company wants to reduce the waste in the cutting process to save money and also sell products of good quality to its customers. The goal is therefore twofold. On one hand the raw material losses after cutting have to be minimized, on a second hand to match customer specifications sold products have to be defect free. This corresponds to solve the $2BP_l$ and $2BP_d$ problems respectively. The following section is split in two parts. One describes $2BP_l$ and the second focuses on $2BP_d$.

1.5.1 Two-dimensional cutting problems with leftover

The $2BP_l$ is important from a practical point of view since it maximizes the residual part of the last cut bin. The main motivation is that the residual part can be reused afterwards. Having a long residual part saves more raw material for future usage. For example it may be used to cut some extra items. In other words since the residual length of the last cut bin has to be maximized this is equivalent to minimize the used length of the last bin. The residual length corresponds to the largest remaining waste strip, the used length refers to the shortest used strip width. This section first focuses on how to handle leftover for the one dimensional case. A more precise description is then given for the 2BP since it is the main interest of this section.

First works on the bin-packing problem with leftover mainly focus on the 1BP. The main approach is to rewrite the ILP model (1.19)-(1.21) for the 1BP by extending it to consider leftover. Leftover is integrated in the objective function and this leads to solve two pricing problems. The first one is exactly the same as the one for the 1BP, the second one is a modified version to consider leftover. Optimal integer solutions are then obtained using the branching schemes explained in Section 1.2.2.5.

For the two dimension case, an initial heuristic approach is proposed by Puchinger et al. [67] considering three-stage patterns. The problem considered

by the authors has a special requirement concerning the output patterns. In their case, items cut in a solution have to respect a certain order. This order is related to wagons collecting cut items and sending them to another production unit in the factory. Authors solve the $2BP_l$ using First-Fit heuristic, heuristic branch-and-bound and a genetic algorithm. From computational tests on real-world instances, their genetic algorithm is the best suited approach to solve their problem.

Meta-heuristics perform well to solve the $2BP_l$ with three-stage patterns. In another work on the subject, Dusberger and Raidl [24] develop heuristics based on the Variable Neighbourhood Search (VNS). Initial heuristics aim to provide an initial solution to the problem. VNS meta-heuristic is based on a ruin-and-recreate principle. Since a $2BP_l$ solution is a set of cutting patterns, authors evaluate with predefined fitness functions the quality of a cutting pattern. If the quality is weak, the cutting pattern is partially destroyed. Then a valid solution is rebuilt using constructive heuristics. Results on standard literature instances demonstrate the efficiency of the approach compared to the exact method of Puchinger and Raidl [66]. In later works presented in Dusberger and Raidl [25], the authors adapt the recreate part using dynamic programming instead of heuristics leading to better results. Experiments are made on modified literature instances. The dynamic programming approach provides better results than the heuristic one but reported computation times are high on average.

Exact methods also exist to solve the $2BP_l$ but are not numerous. One of the contributions on the subject was proposed by Andrade et al. [4]. Starting from the two ILP models described in Lodi et al. [49] for the two-stage 2BP, the authors extend them to deal with bins of multiple size and handle left-overs for all of them. This results in two formulations based on the two ones in Lodi et al. [49] with many extra constraints and variables. Computational experiments were made on randomly generated small instances to compare both formulations. Authors compare them and results show that their second model, inspired from the second one described in Section 1.3.3.1, has the smallest computation time on average. Authors clearly state that their models are not suitable to solve large problem instances.

In an industrial context where item sets have to be cut consecutively, researchers focus on the importance to carry leftovers after cutting a given item set to use them in the cutting process of the next item set. This can be reduced to solve consecutively a cutting problem for each item set to cut. From a practical point of view, using leftovers saves raw material but a large amount of available leftovers increases both handling time and problem complexity. Moreover sometimes next item sets to handle are not known in advance. More details on this type of problem are found in Trkman and Gradisar [77] and Tomat and Gradisar [76]. In Birgin et al. [8], author

1.5.2 Two-dimensional cutting problems with defects

The problem of cutting items considering defects is not an easy task but has a major impact in the industry. For example no one would like to buy a windshield with a bubble in the middle of it for obvious security reasons. In this case a sold product or a cut item has to be defect free. Nevertheless in some industry customers allow defective pieces. Since generic cutting problems with defects have not been studied a lot in the literature, the section first describes the 2KP with defects and then the 2BP with defects.

A first way to solve the $2KP_d$ is to use dynamic programming as done by Hahn [36] and Scheithauer and Terno [74]. The main drawback of the dynamic programming approach is its huge size since each state stores a plate dimension (w, h) and coordinates (x, y). To reduce the number of states, one can only create a heuristic subset of states as outlined by Afsharian et al. [1]. This proves to be efficient to solve the unbounded $2KP_d$. Instead of using dynamic programming, an attempt to solve the $2KP_d$ with heuristic branch-and-bound has been done by Neidlein et al. [60]. The proposed approach is an extension of the one from Morabito and Arenales [58] for the defect free case.

In addition to the $2KP_d$, researchers also focus on the $2BP_d$. The online problem version was considered in De Gelder and Wagelmans [19]. A bin of unlimited width and fixed height is considered. The authors then solve the problem with a two step heuristic. An initial set of patterns is produced until item productions are satisfied. Then patterns are assigned to bins with goal to avoid defects. The idea is to use waste in created patterns and swap items in a pattern to avoid defects. The defect avoiding subroutine was also used by Jin et al. [45] to heuristically solve the $2BP_{sd}$.

Very specific cutting problems from textile, steel and forest industries have also been studied by researchers (see Twisselmann [78], Sarker [73], Ozdamar [63] and Rönnqvist and Åstrand [69]).

1.6 Conclusion

Among the large number of variants of cutting problems, a common solving procedure rises. This one is based on the reformulation of 2BP using Dantzig-Wolfe reformulation. This gives a master problem which has to select cutting patterns commonly tackled with column generation and pricing problems which have to build cutting patterns. Pricing problems handle industrial constraints.

Pricing problems after Dantzig-Wolfe decomposition are a special 2KP with industrial cutting constraints. When 2KP is considered in its unbounded version ($d_i = \infty, \forall i \in \mathcal{I}$), dynamic programming provides the optimal solution regardless of cutting constraints. This approach is also able to tackle large scales instances. When 2KP is bounded, the optimal solution can be obtained

using pseudo-polynomial size ILP models when the number of cutting stages is two or three. For any-stage problem such models do not perform well and implicit enumeration is preferred instead. In many cases, solution space of 2KP variants can be reduced exploiting cutting constraints to eliminate redundant patterns. Nevertheless in a column generation context pricing problems have to be solved very quickly. This limits the complexity of built cutting patterns.

The bottleneck of the column generation is pricing problems. Performing a branch-and-price to solve large scale instances is out of range due to high computation time. Even if exact solving with branch-and-price is prohibited, good quality solutions can be obtained using diving heuristics. When one wants to solve the 2BP to optimality, based 2KP pseudo-polynomial size ILP models have to be used. Best results are achieved using pseudo-polynomial flow formulations. For special cases of the 2BP handling defects and/or left-over, heuristic approaches are preferred since the problem becomes much more harder to solve than the classical 2BP. In the present state of the art, it is hard to solve to optimality large scale instances of 2BP and its variants.

Chapter 2

Knapsack problem

In this chapter, new exact methods for guillotine 2KP are presented. The retained approach is to start from the dynamic program used to solve the unbounded guillotine 2KP. Using the paradigm of Martin et al. [56], hypergraph formalism is introduced and a hypergraph representation of the dynamic program is then detailed. The dynamic program is then written as a max-cost flow problem based on the hypergraph. This flow formulation is then enriched with side constraints to solve the initial guillotine 2KP. In parallel, forward labelling dynamic programming recursion are derived from the hypergraph model. Since the hypergraph is the main support of described methods, dominance rules and filtering procedure based on Lagrangian reduced costs fixing are used to reduce its size. Exact methods are then benchmarked on a set of instances from the literature and on datasets derived from the glass industry. This chapter is partially based on Clautiaux et al. [13].

2.1 A dynamic program for the unbounded 2KP

The dynamic programming is the one of the most effective methods to solve unbounded 2KP as outlined in Section 1.3.2. Idea of such program is to enumerate implicitly the set of all feasible cutting patterns for a given bin. Nevertheless this is possible in practice only when the cutting problem is unbounded. The dynamic program hereinafter is an adaptation of the recursion of Beasley [5]. Furthermore preprocessing techniques are outlined to reduce the number of states of this dynamic program. Since the considered problem here admit item rotation, one needs to handle it. Recall that $\mathcal{I}$ is the initial item set to cut for a 2KP instance. The number of available items is fixed to $m = |\mathcal{I}|$. Let $\mathcal{I}'$ be the set of initial items with rotation based on $\mathcal{I}$ (*i.e.* $\mathcal{I}' = \{i' : w_{i'} = h_i, h_{i'} = w_i, e_{i'} = e_i, \forall i \in \mathcal{I}\}$). Note that there is a correspondence between sets $\mathcal{I}$ and $\mathcal{I}'$. If items are indexed by their positions from 1 to m in $\mathcal{I}$ and $\mathcal{I}'$, the i-th item in $\mathcal{I}$ refers to the initial item, while the i-th item in $\mathcal{I}'$ refers to its rotated version. To be valid a cutting pattern has to

respect cutting constraints and also that the number of times an item and its rotated version are cut does not exceed the demand for item d_i. Finally the set containing all possible items to cut is given by $\bar{\mathcal{I}} = \mathcal{I} \cup \mathcal{I}'$.

2.1.1 Dynamic program for the U-2KP-RE-4-r

According to the guillotine cut and since the problem discussed here is staged, a cut of level $j \in \{2, 3\}$ has to be parallel to an edge and orthogonal to cuts of levels $j-1$ and $j+1$. Note also that each cut length has to be equal to the height or the width of an item in $\mathcal{I}$ since the cut lengths are restricted. First cut is always performed along the width W of the bin.

Let $(w, h)^j$ be the state related to cutting the plate of size (w, h) from stage $j \in \{1, \ldots, 4\}$. Let also $\overline{(w, h)^j}$ be the same state in which it is mandatory to cut an item with the next cut. For a given state s, let $U(s)$ be the maximum profit/utility that can be obtained from this state. The optimal value that can be obtained when a bin of size (W, H) is considered is obtained by computing $U((W, H)^1)$. Let $\mathcal{W}(w, h)$ and $\mathcal{H}(w, h)$ be the set of all possible widths and heights of items in $\bar{\mathcal{I}}$ which fit into rectangle $w \times h$:

$$\mathcal{W}(w,h) = \bigcup_{i \in \bar{\mathcal{I}}: w_i \leqslant w, h_i \leqslant h} \{w_i\}, \quad \mathcal{H}(w,h) = \bigcup_{i \in \bar{\mathcal{I}}: w_i \leqslant w, h_i \leqslant h} \{h_i\}$$

The set of all possible cutting patterns considering four stages of cuts is generated by the following recurrence relations:

$$U((w,h)^1) = \max\left\{0, \max_{w' \in \mathcal{W}(w,h)} \left\{U(\overline{(w',h)^2}) + U((w-w',h)^1)\right\}\right\} \tag{2.1}$$

$$U((w,h)^2) = \max\left\{0, \max_{h' \in \mathcal{H}(w,h)} \left\{U(\overline{(w,h')^3}) + U((w,h-h')^2)\right\}\right\} \tag{2.2}$$

$$U((w,h)^3) = \max\left\{0, \max_{w' \in \mathcal{W}(w,h)} \left\{U(\overline{(w',h)^4}) + U((w-w',h)^3)\right\}\right\} \tag{2.3}$$

$$U(\overline{(w,h)^2}) = \max_{i \in \bar{\mathcal{I}}: w_i = w, h_i \leqslant h} \left\{e_i + U((w,h-h_i)^2)\right\} \tag{2.4}$$

$$U(\overline{(w,h)^3}) = \max_{i \in \bar{\mathcal{I}}: h_i = h, w_i \leqslant w} \left\{e_i + U((w-w_i,h)^3)\right\} \tag{2.5}$$

$$U(\overline{(w,h)^4}) = \max\left\{0, \max_{i \in \bar{\mathcal{I}}: w_i = w, h_i \leqslant h} \left\{e_i + U(\overline{(w,h-h_i)^4})\right\}\right\} \tag{2.6}$$

Note that in a state $(w, h)^j$, it is always possible to transform the plate into waste (as it is modelled by $\max\{0, .\}$). In a state $\overline{(w, h)^j}$, it is mandatory to cut an item first. At stage 4, either one cuts an item, or waste is produced. The state space has a pseudo-polynomial size depending on the size of the considered bin (W, H) and the item set $\bar{\mathcal{I}}$.

2.1.2 Hypergraph representation of the dynamic program

The dynamic program (2.1)-(2.6) allows one to represent the set of all cutting patterns, when the demand of each item is unbounded. According to the paradigm of Martin et al. [56], the search of a maximum profit cutting pattern using this dynamic program is equivalent to the search for a max-cost flow in the corresponding directed acyclic hypergraph with a single sink. This directed acyclic hypergraph is denoted by $G^0 = (\mathcal{V}^0, \mathcal{A}^0)$. The vertex set $\mathcal{V}^0$ is composed of all states from the previous dynamic program but also of so-called *boundary states* that are used for initialization of the recursion and which correspond to single item $i \in \mathcal{I}$ or waste $\varnothing$. These boundary states are the sources of the hypergraph. Its sink t corresponds to state $(W, H)^1$, it stands for the bin. Each hyperarc a has a head set $\mathcal{H}(a)$, which contains a unique vertex, and a tail multiset $\mathcal{T}(a)$, which contains one or more vertices. In the hypergraph corresponding to dynamic program (2.1)-(2.6), every hyperarc tail is a simple set. However in the following part of this manuscript, hypergraph simplification rules will transform the set structure into a multiset. Particularity of multiset is that a vertex can occur more than once. The notation $n_a(v)$ is used to denote the number of times a vertex $v \in \mathcal{V}^0$ is contained in $\mathcal{T}(a), a \in \mathcal{A}^0$. Formally a hyperarc a represents a cutting decision that turns state (*i.e.* plate) $v \in \mathcal{H}(a)$ into states (*i.e.* subplates) in $v \in \mathcal{T}(a)$. The hyperarc set $\mathcal{A}^0$ contains the set of all those cutting decisions. An example of a hypergraph is given in Figure 2.1. When rotation is considered as described here, one needs to create a hyperarc a with boundary state $i \in \mathcal{I}$ in its tail multiset and the corresponding item with rotation $i' \in \mathcal{I}'$ in its head set.

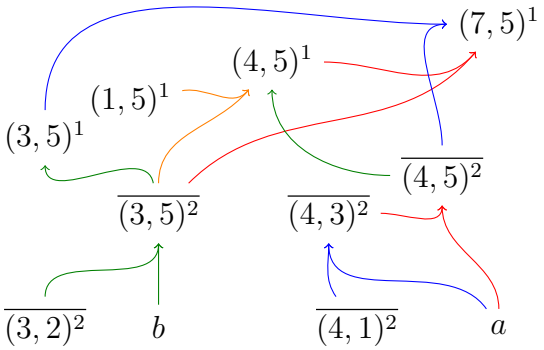

Figure 2.1: The hypergraph related to a bin of size $(7, 5)$ and two items: a of size $(4, 2)$ and b of size $(3, 3)$. Rotations are not permitted and only two cutting stages are allowed, therefore states of level 2 are only $\overline{(w, h)}$ states. The waste vertex and related hyperarcs are not represented.

The dynamic program (2.1)-(2.6) can then be solved to optimality by recursion using Bellman's method. Equivalently, using the hypergraph repre-

sentation, the optimal solution can be obtained by a forward traversal of the hypergraph vertices. Such traversal imposes to cross hypergraph vertices in topological order from its sources to its sink. More explicitly, considering each vertex $v \in \mathcal{V}^0$ in topological order, the maximum cost flow value $U(v)$ is computed using:

$$U(v) = \max_{a \in \Gamma^-(v)} \left\{ \sum_{v' \in \mathcal{T}(a)} n_a(v') U(v') \right\} \tag{2.7}$$

Set $\Gamma^-(v)$ represents the incoming hyperarc set of vertex v. The forward dynamic program is initialized with $U(v) = 0$ for boundary state representing waste and $U(v) = e_i$ for boundary states representing an item. The problem optimal solution is obtained by computing $U(t) = U((W, H)^1)$.

Parallels can be drawn with existing models of 2KP. There is equivalence between the hypergraph and the AND/OR graph defined by Morabito and Arenales [58]. A directed AND/OR graph is defined by its particular arc type. An OR arc is an arc with a head and connected to an intermediate vertex. An AND arc is an connection between this intermediate vertex and a set of vertices. Clearly this is equivalent to the previous defined notion of hyperarc, one needs to remove the intermediate vertex.

With respect to a more general definition of Berge [6], a hyperarc a has a head multiset and a tail multiset containing one or more vertices (*i.e.* $|\mathcal{H}(a)| \geqslant 1$ and $|\mathcal{T}(a)| \geqslant 1$). In the hypergraph definition retained by Gallo et al. [32], authors distinguished two types of hyperarcs. The first ones are called backward hyperarcs (B-arcs) and have the particularity to contain only a single vertex in their head set (*i.e.* $|\mathcal{H}(a)| = 1$ and $|\mathcal{T}(a)| \geqslant 1$). The second type of defined hyerarcs are called forward hyperarcs (F-arcs) and contained only a single vertex in their tail set (*i.e.* $|\mathcal{H}(a)| \geqslant 1$ and $|\mathcal{T}(a)| = 1$). From definition of Gallo et al. [32], the AND/OR graph notation of Morabito and Arenales [58] is equivalent to the one of F-hypergraph, a hypergraph composed of only forward hyperarcs. The notation retained through the rest of this manuscript will be a B-hypergraph, a hypergraph composed of only backward hyperarcs. A representation of hyperarcs is given in Figure 2.2.

There is a second equivalence between the hypergraph representation and enumerative methods proposed by Silva et al. [75] and Furini et al. [30]. Each author tackled their respective problem by means of a pseudo-polynomial size ILP model in which each variable is related to a way to cut a plate into subplates. A graphical representation of such cutting decision is related to use of a hyperarc. A plate is divided into two subplates which matches the previous definition of a hyperarc. Therefore the set of all cutting decisions is represented by a hypergraph but none of the authors used this formalism.

Through the rest of this manuscript and using definitions of Gallo et al. [32], the proposed representation of the unbounded 2KP is achieved by means

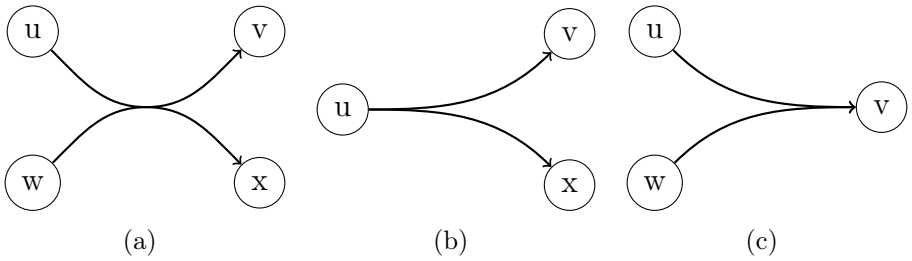

Figure 2.2: Different type of hyperarcs: generic hyperarc (a) from Berge [6], forward hyperarc (b) from Gallo et al. [32] and used by Morabito and Arenales [58], backward hyperarc (c) from Gallo et al. [32]

of a B-hypergraph, to match with the paradigm of Martin et al. [56]. To make reading easier, the term hypergraph will be used to refer to the proposed B-hypergraph representation instead. The hypergraph formalism is also preferred since it allows one to use classical network-flow models.

2.1.3 Hypergraph preprocessing

The hypergraph size determines the number of operations needed to solve the corresponding dynamic program. Therefore reducing its size decreases the dynamic program solution time. The following simplification rules help to reduce the hypergraph size.

2.1.3.1 Simple pattern enumeration

The following rule is dedicated only when restricted cuts are used. The purpose is to first reformulate recurrence relation (2.6) since its models a one-dimensional knapsack problem. The latter admits many symmetric solutions associated to different permutation of the position of items in a strip. A way to remove such symmetries is to enumerate all possible solutions of the associated one-dimensional knapsack problem. A hyperarc is added in the hypergraph for each of these solutions. Since the proposed dynamic program models the unbounded 2KP, this solution enumeration allows to enforce the bound d_i of each item $i \in \mathcal{I}$.

Let $\mathcal{KP}(\overline{(w,h)^4})$ be the set of all one-dimensional knapsack solutions related to a plate $\overline{(w,h)^4}$ and using items in $\bar{\mathcal{I}}$ such that the bound d_i on each item $i \in \mathcal{I}$ is not exceeded. Each solution is denoted by an item set $\hat{\mathcal{I}} \in \mathcal{KP}(\overline{(w,h)^4})$, where $\hat{\mathcal{I}}$ represents the set of items used in this solution. Hence, the recurrence relation (2.6) can be rewritten as:

$$U(\overline{(w,h)^4}) = \max_{\hat{\mathcal{I}} \in \mathcal{KP}(\overline{(w,h)^4})} \left\{ \sum_{i \in \hat{\mathcal{I}}} e_i \right\} \tag{2.8}$$

When applying enumeration at the last cutting stage, the hyperarc a associated with the cutting decision has a special structure regarding $\mathcal{T}(a)$. In Section 2.1.2, $\mathcal{T}(a)$ has been defined as a multiset instead of a set. This definition is related to the special structure of $\mathcal{T}(a)$ obtained after using the rewriting of (2.6) with (2.8). Indeed using an enumerated one-dimensional knapsack solution in $\mathcal{KP}(\overline{(w,h)^4})$, an item boundary state can occur more than once. For instance one can cut from a state $\overline{(w,h)^4}$, w times an item of size $(1,h)$. Therefore the definition of $\mathcal{T}(a)$ as a set does not hold since a vertex v can be contained more than one time. The definition of multiset allows this type of behaviour. The previous defined notation $n_a(v)$ gives the number of times a vertex v occurs in $\mathcal{T}(a)$.

Remark that its is not mandatory to enumerate solutions for all possible states $\overline{(w,h)^4}$. From restricted and exact cuts in the considered problem here, the maximum width of states $\overline{(w,h)^4}$ is bounded by $w_{max} = \max_{i \in \bar{\mathcal{I}}} \{w_i\}$. In the same way only positions $h = h_i, i \in \bar{\mathcal{I}}$ are relevant.

2.1.3.2 Symmetry breaking

A simplification to apply on the hypergraph is to remove symmetry between cutting patterns. Since the dynamic program enumerates all possible cutting patterns, some of them can be equivalent. According to Valério de Carvalho [79] for the one-dimensional case, two cutting patterns c and c' are symmetric if c is obtained from c' by a sequence of swapping of same stage strips. An example of two symmetric patterns is depicted in Figure 2.3.

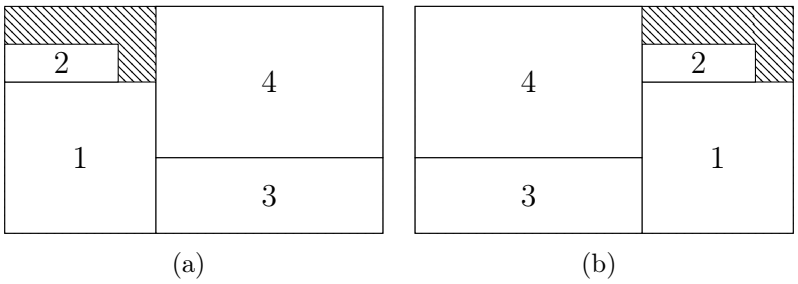

Figure 2.3: Representation of two equivalent patterns

This type of symmetry can be partially removed from the hypergraph. Practically speaking for each vertex v related to state $(w,h)^j$, a value $l_v \in \mathbb{N}^+$ is stored, which represents the largest cut length that produced this state starting

from an equivalent stage vertex. Then we can remove hyperarcs $\Gamma^-(v)$ related to a cut of length greater than l_v. Although this technique does not exclude all possible symmetries, it reduces significantly the size of the hypergraph.

If one is interested to remove all possible symmetries between patterns at a given cutting stage, the dynamic program has to be rewritten. Each state is enriched with a new attribute l referring to the maximum cut length which can be performed on this vertex. For instance each state $(w,h)^1$ will be of the form $(w,h,l)^1$ and obtained as follows:

$$U((w,h,l)^1) = \max\left\{0, \max_{w' \in \mathcal{W}(w,h): w' \leqslant l} \left\{U(\overline{(w',h)^2}) + U((w-w',h,w')^1)\right\}\right\} \tag{2.9}$$

This new recurrence relation completely removes symmetries for the first cutting stage. Nevertheless an extra dimension is added to the dynamic program and this results in an increasing of its size. Removing all symmetries does not seem pertinent since the hypergraph size want to be reduced. One will prefer the partial symmetry breaking instead since it is easier to use in practice.

2.1.3.3 Simple plate reduction

A way to reduce the hypergraph size is to merge states that are associated with the same set of cutting patterns. Idea is to use all possible cut length combinations and to show that some positions $w \in \{0,\ldots,W\}$ and $h \in \{0,\ldots,H\}$ are not reachable. The purpose is to decrease the number of vertices as well as the number of hyperarcs in the hypergraph. This technique is largely inspired of the one developed by Christofides and Whitlock [10] as detailed in Section 1.3.2.

Let $\mathcal{W}_c$ (resp. $\mathcal{H}_c$) be the set of all cut length combinations which can be applied up to W (resp. H):

$$\mathcal{W}_c = \left\{ \begin{array}{l} w | w = \sum_{o=1}^{m} w_{\mathcal{I}(o)}\alpha_o + w_{\mathcal{I}'(o)}\beta_o : \\ 1 \leqslant w < W, \alpha_o + \beta_o \leqslant d_{\mathcal{I}(o)}, \\ \alpha_o \in \mathbb{N}, \beta_o \in \mathbb{N}, o = \{1,\ldots,m\} \end{array} \right\}$$

$$\mathcal{H}_c = \left\{ \begin{array}{l} h | h = \sum_{o=1}^{m} h_{\mathcal{I}(o)}\alpha_o + h_{\mathcal{I}'(o)}\beta_o : \\ 1 \leqslant h < H, \alpha_o + \beta_o \leqslant d_{\mathcal{I}(o)}, \\ \alpha_o \in \mathbb{N}, \beta_o \in \mathbb{N}, o = \{1,\ldots,m\} \end{array} \right\}$$

Notation $\mathcal{I}(o)$ and $\mathcal{I}'(o)$ gives the item at index $o \in \{1,\ldots,m\}$ in sets $\mathcal{I}$ and $\mathcal{I}'$. If item rotation is not allowed, $\mathcal{I}'$ and β_o values have to be omitted.

Let $p(w)$ (resp. $q(h)$) be the width (resp. height) nearest from below to w (resp. h) in the new set $\mathcal{W}_c$ (resp. $\mathcal{H}_c$):

$$p(w) = \max\{0, w' | w' \leqslant w, w' \in \mathcal{W}_c\}, w < W$$
$$q(h) = \max\{0, h' | h' \leqslant h, h' \in \mathcal{H}_c\}, h < H$$

Therefore one can reduce the number of states in the dynamic program (2.1)-(2.6) by using $p(w)$ instead of w and $p(h)$ instead of h:

$$U((w,h)^1) = \max\left\{0, \max_{w' \in \mathcal{W}(w,h)} \left\{U(\overline{(w', q(h))^2}) + U((p(w-w'), q(h))^1)\right\}\right\} \tag{2.10}$$

$$U((w,h)^2) = \max\left\{0, \max_{h' \in \mathcal{H}(w,h)} \left\{U(\overline{(p(w), h')^3}) + U((p(w), q(h-h'))^2)\right\}\right\} \tag{2.11}$$

$$U((w,h)^3) = \max\left\{0, \max_{w' \in \mathcal{W}(w,h)} \left\{U(\overline{(w', q(h))^4}) + U((p(w-w'), q(h))^3)\right\}\right\} \tag{2.12}$$

$$U(\overline{(w,h)^2}) = \max_{i \in \bar{\mathcal{I}}: w_i = w, h_i \leqslant h} \left\{e_i + U((p(w), q(h-h_i))^2)\right\} \tag{2.13}$$

$$U(\overline{(w,h)^3}) = \max_{i \in \bar{\mathcal{I}}: h_i = h, w_i \leqslant w} \left\{e_i + U((p(w-w_i), q(h))^3)\right\} \tag{2.14}$$

$$U(\overline{(w,h)^4}) = \max\left\{0, \max_{i \in \bar{\mathcal{I}}: w_i = w, h_i \leqslant h} \left\{e_i + U(\overline{(p(w), q(h-h_i))^4})\right\}\right\} \tag{2.15}$$

2.1.3.4 Enhanced plate reduction

Simplification based on cut length described in Section 2.1.3.3 can be extended to handle more precisely item dimensions and bounds. This implies to redefine sets $\mathcal{W}_c$ and $\mathcal{H}_c$. Idea is to compute for each $w' \in \mathcal{W}(W,H)$ (resp. $h' \in \mathcal{H}(W,H)$) the set $\mathcal{H}_c(w)$ (resp. $\mathcal{W}_c(h)$) of possible cut length combinations using only item $i \in \bar{\mathcal{I}}$ with a width $w_i \leqslant w'$ (resp. a height $h_i \leqslant h'$). This gives rise to the two new sets:

$$\mathcal{W}_c(h) = \left\{ \begin{array}{l} w | w = \sum_{o=1}^{m} w_{\mathcal{I}(o)}\alpha_o + w_{\mathcal{I}'(o)}\beta_o : \\ h_{\mathcal{I}(o)} > h \Rightarrow \alpha_o = 0, \\ h_{\mathcal{I}'(o)} > h \Rightarrow \beta_o = 0, \\ 1 \leqslant w < W, \alpha_o + \beta_o \leqslant d_{\mathcal{I}(o)}, \\ \alpha_o \in \mathbb{N}, \beta_o \in \mathbb{N}, o = \{1, \ldots, m\} \end{array} \right\}$$

$$\mathcal{H}_c(w) = \left\{ \begin{array}{l} h | h = \sum_{o=1}^{m} h_{\mathcal{I}(o)}\alpha_o + h_{\mathcal{I}'(o)}\beta_o : \\ w_{\mathcal{I}(o)} > w \Rightarrow \alpha_o = 0, \\ w_{\mathcal{I}'(o)} > w \Rightarrow \beta_o = 0, \\ 1 \leqslant h < H, \alpha_o + \beta_o \leqslant d_{\mathcal{I}(o)}, \\ \alpha_o \in \mathbb{N}, \beta_o \in \mathbb{N}, o = \{1, \ldots, m\} \end{array} \right\}$$

Let define two new functions $p(w,h)$ (resp. $q(h,w)$) storing the width (resp. height) nearest from below to w (resp. h) in the new set $\mathcal{W}_c(h)$ (resp. $\mathcal{H}_c(w)$):

$$p(w,h) = \max\left\{0, w' | w' \leqslant w, w' \in \mathcal{W}_c(h)\right\}, w < W$$
$$q(h,w) = \max\left\{0, h' | h' \leqslant h, h' \in \mathcal{H}_c(w)\right\}, h < H$$

Let also be $\bar{w}$ and $\bar{h}$ storing the item width and item height nearest to w and h in $\mathcal{W}(W,H)$ and $\mathcal{H}(W,H)$:

$$\bar{w}(w) = \max\left\{0, w' | w' \in \mathcal{W}(W,H)\right\}, w < W$$
$$\bar{h}(h) = \max\left\{0, h' | h' \in \mathcal{H}(W,H)\right\}, h < H$$

Using these new definitions the dynamic program (2.1)-(2.6) is now written as follows:

$$U((w,h)^1) = \max\left\{0, \max_{w' \in \mathcal{W}(w,h)} \left\{ \begin{array}{l} U(\overline{(w', q(h,w'))^2}) + \\ U((p(w-w', \bar{h}(h)), q(h, \bar{w}(w-w')))^1) \end{array} \right\}\right\} \tag{2.16}$$

$$U((w,h)^2) = \max\left\{0, \max_{h' \in \mathcal{H}(w,h)} \left\{ \begin{array}{l} U(\overline{(p(w,h'), h')^3}) + \\ U((p(w, \bar{h}(h-h')), q(h-h', \bar{w}(w)))^2) \end{array} \right\}\right\} \tag{2.17}$$

$$U((w,h)^3) = \max\left\{0, \max_{w' \in \mathcal{W}(w,h)} \left\{ \begin{array}{l} U(\overline{(w', q(h,w'))^4}) + \\ U((p(w-w', \bar{h}(h)), q(h, \bar{w}(w-w')))^3) \end{array} \right\}\right\} \tag{2.18}$$

$$U(\overline{(w,h)^2}) = \max_{i \in \bar{\mathcal{I}}: w_i = w, h_i \leqslant h} \left\{e_i + U((p(w_i, \bar{h}(h-h_i)), q(h-h_i, w_i)^2))\right\} \tag{2.19}$$

$$U(\overline{(w,h)^3}) = \max_{i \in \bar{\mathcal{I}}: h_i = h, w_i \leqslant w} \left\{e_i + U((p(w-w_i, h_i), q(h_i, \bar{w}(w-w_i)))^3)\right\} \tag{2.20}$$

$$U(\overline{(w,h)^4}) = \max\left\{0, \max_{i \in \bar{\mathcal{I}}: w_i = w, h_i \leqslant h} \left\{e_i + U(\overline{(w_i, q(h-h_i, w_i))^4})\right\}\right\} \tag{2.21}$$

The drawback of this simplification rule is that all presented cut length sets should be computed before creating the hypergraph. This increases hypergraph building time.

2.1.3.5 Hypergraph simplification

The following hypergraph simplification rule is generic, as does not rely on the structure of the problem. The proposed reduction is inspired from the vertex and edge contraction used in graphs. This reduction aims to compact vertex and hyperarc based on the following observation. One vertex which has only

one incoming hyperarc can be removed from the hypergraph without any loss of information, as illustrated in Figure 2.4. Let v be a vertex in $\mathcal{V}^0$. If v has only one incoming hyperarc a, v and a are deleted from $\mathcal{V}^0$ and $\mathcal{A}^0$. Then, the tail multiset $\mathcal{T}(a)$ is added to the tail multiset $\mathcal{T}(a')$ for all outgoing hyperarcs $a' \in \Gamma^+(v)$, where $\Gamma^+(v)$ represents the outgoing hyperarc set of vertex v. The total number of tails may increase if v has several incoming hyperarcs.

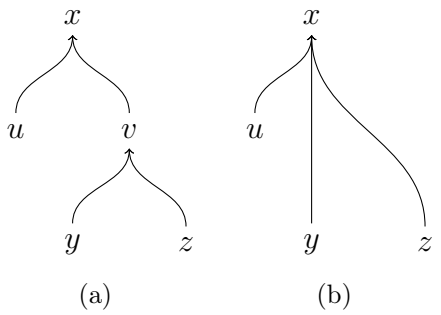

Figure 2.4: Hypergraph before (a) and after (b) vertex contraction. Vertex x has only one incoming hyperarc and then is deleted.

2.1.3.6 Enhanced pattern enumeration

The pattern enumeration at the last cutting stage described in Section 2.1.3.1 is simple since its implies to solve small 1KP instances. This idea can be generalized to enumerate partial patterns when building the hypergraph. Main advantage of the complete enumeration of patterns is that it takes into account the item bound constraints, but the computational cost is huge. On the other hand, the dynamic program has a reasonable computational cost, but it does not take into account the bound constraints. The proposed idea is to mix both approaches by replacing some part of the dynamic program by partially enumerated patterns.

This partial enumeration is implemented using *meta-items*, each one representing a partial vertical or horizontal stack of item copies satisfying item bounds. When restricted states $\overline{(w,h)^j}, j \in \{2,3,4\}$ are considered, instead of choosing one item to initiate the strip, a meta-item (or equivalently the items that it represents) is selected instead. There is potentially an exponential number of possible meta-items to initiate a given strip. To restrict this number an additional parameter δ is introduced.

Formally, let α_i^r (resp. β_i^r) be the number of copies of item $i \in \mathcal{I}$ (resp. $i \in \mathcal{I}'$) included into meta-item r. Given three values $0 < w \leqslant W$, $0 < h \leqslant H$, and $0 < \delta \leqslant \min_{i \in \bar{\mathcal{I}}} \{w_i\}$, the following set $\mathcal{R}^x(w,h,\delta)$ of vertical meta-items is defined. Each meta-item $r \in \mathcal{MI}^x(w,h,\delta)$ forms in the cutting pattern a partial vertical stack of width w containing copies of items $i \in \mathcal{I}$ and copies of

items $i \in \mathcal{I}'$ such that $w - \delta < w_i \leqslant w$ and $h_i \leqslant h$. Items that do not belong to r may only be cut in other vertical stacks or in the same stack above the item copies in r. This is formally given by the following definition:

$$r \in \mathcal{MI}^x(w,h,\delta) \Leftrightarrow \begin{cases} (\exists i \in \mathcal{I}, w_i = w : \alpha_i^r > 0) \bigvee (\exists i \in \mathcal{I}', w_i = w : \beta_i^r), \\ \alpha_i^r > 0 \Rightarrow w - \delta < w_i \leqslant w \text{ and } h_i \leqslant h, \quad \forall i \in \mathcal{I}, \\ \beta_i^r > 0 \Rightarrow w - \delta < w_i \leqslant w \text{ and } h_i \leqslant h, \quad \forall i \in \mathcal{I}', \\ \alpha_{\mathcal{I}(o)}^r + \beta_{\mathcal{I}'(o)}^r \leqslant d_{\mathcal{I}(o)}, \quad o = \{1,\dots,m\} \\ \sum_{o=1}^m \alpha_{\mathcal{I}(o)}^r h_{\mathcal{I}(o)} + \beta_{\mathcal{I}'(o)}^r h_{\mathcal{I}'(o)} \leqslant H \end{cases}$$

The first condition ensures that one item has width w (and thus a restricted pattern is built). The second and third conditions ensure that the size of the items in the meta-item satisfies the requested limitations. The fourth condition ensures that the meta-item satisfies item bound constraints. The last condition ensures that the meta-item height does not exceed the plate height.

Analogously, given values $0 < w \leqslant W$, $0 < h \leqslant H$, and $0 < \delta \leqslant \min_{i \in \bar{\mathcal{I}}} \{h_i\}$, we define the following set $\mathcal{MI}^y(h,w,\delta)$ of horizontal meta-items:

$$r \in \mathcal{MI}^y(h,w,\delta) \Leftrightarrow \begin{cases} (\exists i \in \mathcal{I}, h_i = h : \alpha_i^r > 0) \bigvee (\exists i \in \mathcal{I}', h_i = h : \beta_i^r > 0), \\ \alpha_i^r > 0 \Rightarrow h - \delta < h_i \leqslant h \text{ and } w_i \leqslant w, \quad \forall i \in \mathcal{I}, \\ \beta_i^r > 0 \Rightarrow h - \delta < h_i \leqslant h \text{ and } w_i \leqslant w, \quad \forall i \in \mathcal{I}', \\ \alpha_{\mathcal{I}(o)}^r + \beta_{\mathcal{I}'(o)}^r \leqslant d_{\mathcal{I}(o)}, \quad o = \{1,\dots,m\} \\ \sum_{o=1}^m \alpha_{\mathcal{I}(o)}^r w_{\mathcal{I}(o)} + \beta_{\mathcal{I}'(o)}^r w_{\mathcal{I}'(o)} \leqslant W \end{cases}$$

Let $\mathcal{MI}^x$ (resp. $\mathcal{MI}^y$) be the whole set of vertical (resp. horizontal) meta-items:

$$\mathcal{MI}^x = \bigcup_{\substack{w = \{1,\dots,W\}, \\ h = \{1,\dots,H\}, \\ \delta \in \mathbb{N}}} \mathcal{MI}^x(w,h,\delta)$$

$$\mathcal{MI}^y = \bigcup_{\substack{w = \{1,\dots,W\}, \\ h = \{1,\dots,H\}, \\ \delta \in \mathbb{N}}} \mathcal{MI}^y(w,h,\delta)$$

For a meta-item r, one can obtained its total value $\bar{p}_r$, total height $\bar{h}_r$ and/or total width $\bar{w}_r$:

$$\bar{p}_r = \sum_{o=1}^{m} \alpha^r_{\mathcal{I}(o)} p_{\mathcal{I}(o)} + \beta^r_{\mathcal{I}'(o)} p_{\mathcal{I}'(o)}, \qquad \forall r \in \mathcal{MI}^x \cup \mathcal{MI}^y$$

$$\bar{h}_r = \sum_{o=1}^{m} \alpha^r_{\mathcal{I}(o)} h_{\mathcal{I}(o)} + \beta^r_{\mathcal{I}'(o)} h_{\mathcal{I}'(o)}, \qquad \forall r \in \mathcal{MI}^x$$

$$\bar{w}_r = \sum_{o=1}^{m} \alpha^r_{\mathcal{I}(o)} w_{\mathcal{I}(o)} + \beta^r_{\mathcal{I}'(o)} w_{\mathcal{I}'(o)}, \qquad \forall r \in \mathcal{MI}^y$$

Note that, by definition, $\mathcal{MI}^x(w,h,\delta) = \varnothing$ if $w \notin \mathcal{W}(W,H)$, and $\mathcal{MI}^y(h,w,\delta) = \varnothing$ if $h \notin \mathcal{H}(W,H)$. Suppose now that for each $w \in \mathcal{W}(W,H)$ a value δ_w, $0 \leqslant \delta_w \leqslant \min_{i\in\bar{\mathcal{I}}}\{w_i\}$, is fixed, and for each $h \in \mathcal{H}(W,H)$ a value δ_h, $0 \leqslant \delta_h \leqslant \min_{i\in\bar{\mathcal{I}}}\{h_i\}$, is fixed. Then the recursive formulae for states $\overline{(w,h)^k}$ can be rewritten in the following way without loss of any proper patterns from the set of feasible solutions:

$$U(\overline{(w,h)^2}) = \begin{cases} \max\limits_{r\in\mathcal{MI}^x(w,h,\delta_w):\bar{h}_r\leqslant h} \left\{\bar{p}_r + U((w,h-\bar{h}_r)^2)\right\}, & \text{if } \delta_w > 0, \\ \max\limits_{i\in\bar{\mathcal{I}}:w_i=w,h_i\leqslant h} \left\{e_i + U((w,h-h_i)^2)\right\}, & \text{if } \delta_w = 0, \end{cases}$$

$$U(\overline{(w,h)^3}) = \begin{cases} \max\limits_{r\in\mathcal{MI}^y(h,w-\delta_w,\delta_h):\bar{w}_r\leqslant w} \left\{\bar{p}_r + U((w-\bar{w}_r,h)^3)\right\}, & \text{if } \delta_h > 0, \\ \max\limits_{i\in\bar{\mathcal{I}}:h_i=h,w_i\leqslant w-\delta_w} \left\{e_i + U((w-w_i,h)^3)\right\}, & \text{if } \delta_h = 0, \end{cases}$$

$$U(\overline{(w,h)^4}) = \begin{cases} \max\left\{0, \max\limits_{r\in\mathcal{MI}^x(w,h-\delta_h,1)}\{\bar{p}_r\}\right\}, & \text{if } \delta_w > 0, \\ \max\left\{0, \max\limits_{i\in\bar{\mathcal{I}}:w_i=w,h_i\leqslant h-\delta_h}\left\{e_i + U(\overline{(w,h-h_i)^4})\right\}\right\} & \text{if } \delta_w = 0, \end{cases}$$

If all values δ are fixed to zero, the modified dynamic program reduces to the original one presented in Section 2.1.1. The larger the values δ are, the larger is the number of meta-items. One needs to find a trade-off between the complexity (or the size) of the dynamic program and the strength of the approximation of the space of valid cutting patterns by the space of feasible solutions of the dynamic program. This trade-off is parametrized by defining thresholds $\Delta^{size} \geqslant 0$ on the size of the sets of meta-items and $\Delta^{\text{diff}} \geqslant 0$ on values δ, respectively for dimension w and h. Given these thresholds, values δ are determined the following way. For each $w \in \mathcal{W}(W,H)$, δ_w is the largest value $\delta \leqslant \min\left\{\Delta^{\text{diff}}_w, \min_{i\in\bar{\mathcal{I}}}\{w_i\}\right\}$ such that $\mid \mathcal{MI}^x(w,H,\delta) \mid \leqslant \Delta^{size}_w$. As $\mathcal{W}^x(w,H,0) = \varnothing$ for any w, such value δ always exists. Analogously, for each $h \in \mathcal{H}(W,H)$, δ_h is the largest value $\delta \leqslant \min\left\{\Delta^{\text{diff}}_h, \min_{i\in\bar{\mathcal{I}}}\{h_i\}\right\}$ such that $\mid \mathcal{MI}^y(h,W,\delta) \mid \leqslant \Delta^{size}_h$. Note again that $\mathcal{W}^y(h,W,0) = \varnothing$ for any h. The sets of meta-items are computed by enumeration.

To illustrate such generation of meta-items, consider an instance with a bin of size $(4, 3)$ and three items $1 \times i_1 = (4, 1), 2 \times i_2 = (3, 1), 1 \times i_3 = (2, 1)$. Set $\mathcal{MI}^x(4, 3, \delta)$ is computed with $\delta = 2$. Initially meta-items composed of only items i_1 are created (see Figure 2.5(b)). Secondly extra items are added to meta-items derived from item i_1. Since $\delta = 2$, only item i_2 is added (see Figure 2.5(c) and Figure 2.5(d)). From each created meta-item, a valid cutting pattern can be obtained (see Figure 2.6). At this point, item i_3 is not cut because $\delta = 2$. If the cutting process continues with horizontal cuts it will be possible to cut item i_3. A representation of the resulting patterns is given in Figure 2.7. Note that all meta-items containing item i_2 are created, therefore it is impossible to add it again to patterns of Figure 2.6(a) and Figure 2.6(b).

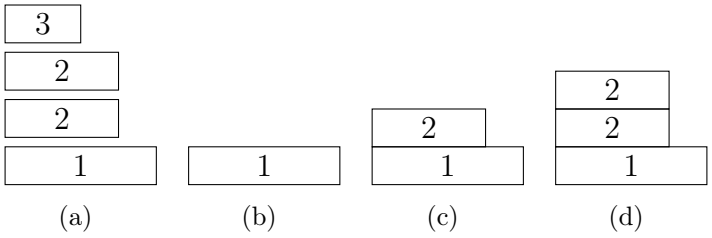

Figure 2.5: Vertical meta-items generation for three items: initial items (a), meta-items obtained with item i_1 only ($\delta = 1$) (b), extra meta-item obtained by adding one copy of item i_2 to meta-item in composed of item i_1 ($\delta = 2$) (c), extra meta-item obtained by adding two copies of item i_2 to meta-item composed of item i_1 ($\delta = 2$) (d)

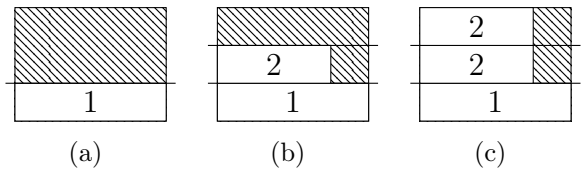

Figure 2.6: Vertical cutting patterns for a bin $(4, 3)$ using meta-items (b),(c) and (d) from Figure 2.5

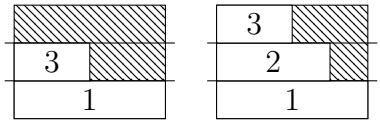

Figure 2.7: Complete cutting patterns for a bin $(4, 3)$ using vertical patterns from Figure 2.6 and adding missing item i_3. Since all meta-items containing item i_2 are created, it is impossible to add it again to patterns from Figure 2.6(a) and Figure 2.6(b)

2.2 A direct ILP formulation for the bounded 2KP

As proposed by Martin et al. [56], the hypergraph representation underlying the dynamic programming recursion can give rise to an ILP flow-model. This ILP model can be augmented with side-constraints which allow to enforce item upper bounds if needed. A solution to the dynamic program is a selection of hyperarcs that gives rise to the maximum value in Bellman's recursive formula (2.7). This selection of hyperarcs forms a directed acyclic hypergraph. Equivalently this combinatorial structure can be identified as the set of hyperarcs carrying a flow of max-cost value into the sink node. Therefore solving the dynamic program is equivalent to solve a max-cost flow problem in this hypergraph. One can derive an ILP formulation for this flow problem. However the formulation has pseudo-polynomial size as is the size of the hypergraph.

The formulation is in terms of integer variables x_a representing the flow value going through hyperarc $a \in \mathcal{A}^0$. Let $\mathcal{A}^0(i)$ be the multiset of hyperarcs whose tail sets include a boundary vertex representing item $i \in \mathcal{I}$. Note that since $\mathcal{T}(a)$ is a multiset, one needs to know the number of times (multiplicity) $n_a(v)$ vertex $v \in \mathcal{V}^0$ is cut when choosing hyperarc $a \in \mathcal{A}^0$. The vector of variables x_a, $a \in \mathcal{A}^0$ is denoted by $\mathbf{x}$. The ILP formulation takes the form:

$$\max \sum_{i \in \mathcal{I}} e_i \sum_{a \in \mathcal{A}^0(i)} n_a(i) x_a \tag{2.22}$$

$$\text{s.t.} \sum_{a \in \Gamma^-(v)} x_a - \sum_{a' \in \Gamma^+(v)} n_{a'}(v) x_{a'} = 0, \quad \forall v \in \mathcal{V}^0 \backslash \{t \cup \mathcal{I} \cup \varnothing\} \tag{2.23}$$

$$\sum_{a \in \Gamma^-(t)} x_a = 1 \tag{2.24}$$

$$x_a \in \mathbb{N}, \quad \forall a \in \mathcal{A}^0 \tag{2.25}$$

Objective function (2.22) aims to maximize the total profit of the selected items. Constraints (2.23) are classical flow conservation constraints. They

ensure that a valid pattern is built. Constraint (2.24) ensures that the total flow coming to the sink vertex t is one and thus that only one plate is used. Based on the results of Martin et al. [56], if $\mathbf{x}$ variables are unbounded, all extreme points of (2.22)-(2.25) are integer. This implies that integrality restriction of $\mathbf{x}$ variables can be relaxed. Note that the flow value going through hyperarcs although integer can be larger than one as illustrated in Figure 2.8.

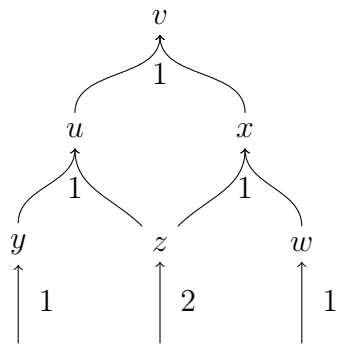

Figure 2.8: Representation of flow values going through hyperarcs

The ILP above can be adapted for the C-2KP-RE-4-r by enforcing item upper bounds, adding the following constraints:

$$\sum_{a \in \mathcal{A}^0(i)} n_a(i) x_a \leqslant d_i, \quad \forall i \in \mathcal{I} \tag{2.26}$$

Constraint set (2.26) limits the number of items which is possible to cut (*i.e.* the sum of flow values going through each hyperarc in $\mathcal{A}^0(i)$ that cover an item $i \in \mathcal{I}$ should not exceed the item upper bound d_i). Such bound enforcement constraints were also used in Valério de Carvalho [79] and Macedo et al. [52] for the cutting stock problem. Note that when variables $\mathbf{x}$ are unbounded by adding side-constraints to the model, the integrality of the LP model relaxation optimal solution is not guaranteed anymore, and therefore one needs to tackle the integer problem which is much harder to solve.

2.3 Lagrangian filtering

The size of the ILP model (2.22)-(2.26) grows too large to be solved directly by commercial solver as soon as one gets on realistic instances. Simplifications rules described in Section 2.1.3 aim to reduce the hypergraph by enforcing item bounds or using cut length properties. Another way to reduce it is to use so-called Lagrangian cost filtering (or simply filtering). The procedure aims to fix a large number of variables to zero by proving that they cannot be part of an optimal solution. In the approach presented here, purpose will be

to remove a large number of variables by fixing them to zero in formulation (2.22)-(2.26). Since each variable carries the flow going through a hyperarc, a variable fixed to zero implies that the variable will not be part of an optimal solution. Consequently the associated hyperarc will not be part of any optimal solution and can be safely removed from the hypergraph.

The Lagrangian cost filtering technique has proved to be a key asset in solving routing or scheduling problems, when using (constrained) path problems as Lagrangian subproblems (see Irnich et al. [44] and Detienne et al. [22]). To introduce the technique, a simple case of a directed acyclic graph is first described. The methodology is then extended to handle hypergraph formalism. As filtering relies on Lagrangian multipliers, several techniques to obtain those multipliers are also detailed.

2.3.1 Standard resource constrained longest path problem

The Resource Constrained Longest Path Problem (RCLPP) entails finding a path from a source s to a sink t in a directed acyclic graph $G = (\mathcal{V}, \mathcal{A})$ with maximum cost, while obeying a threshold constraint on the cumulative resource consumption. Considering R resources, such cumulative consumption to not exceed is given by a vector $\mathcal{R} = (\mathcal{R}^1, \ldots, \mathcal{R}^R)$. Let $c_a = (c_a^1, \ldots, c_a^r)$ be the resource consumption vector of arc $a \in \mathcal{A}$ (where c_a^r represents the amount of resource r consumed by arc a), while p_a is its cost. Using binary variables $x_a = 1$ if the arc $a \in \mathcal{A}$ is in the solution, 0 otherwise, an ILP formulation to the RCLPP is given by:

$$\max \sum_{a \in \mathcal{A}} p_a x_a \tag{2.27}$$

$$\text{s.t.} \sum_{a \in \Gamma^-(v)} x_a - \sum_{a \in \Gamma^+(v)} x_a = 0, \quad \forall v \in \mathcal{V} \backslash \{s \cup t\} \tag{2.28}$$

$$\sum_{a \in \Gamma^-(t)} x_a = 1 \tag{2.29}$$

$$\sum_{a \in \mathcal{A}} c_a^r x_a \leqslant \mathcal{R}^r, \quad r \in \{1, \ldots, R\} \tag{2.30}$$

$$x_a \in \{0, 1\}, \quad \forall a \in \mathcal{A} \tag{2.31}$$

Observe the similarity of the above RCLPP formulation and our formulation given in (2.22)-(2.26).

Filtering for the RCLPP is performed as follows. One applies a Lagrangian relaxation of the resource constraints (2.30) with Lagrangian multipliers π and then derives the associated Lagrangian bound $L(\pi)$ by solving the resulting longest path problem:

$$\tilde{L}(\pi) = \max \left\{ \sum_{a \in \mathcal{A}} (p_a - \sum_{r=1}^{R} \pi_r c_a^r) x_a + \sum_{r=1}^{R} \pi_r \mathcal{R}^r \text{ s.t. (2.28)-(2.29) and (2.31)} \right\}$$

The Lagrangian dual problem consists in adjusting the Lagrangian multipliers π to get the tightest Lagrangian bound $L(\pi)$ by solving $\min_\pi L(\pi)$. This can be done approximatively using for instance a subgradient approach. At each iteration of such subgradient algorithm, one can perform filtering to remove arcs from the graph G. Observe that the longest path solution that yields $L(\pi)$ defines a unit flow from source s to sink t in the directed acyclic graph $G = (\mathcal{V}, \mathcal{A})$. The flow value going through an arc is simply zero or one.

The above longest path problem can be solved by a forward labelling algorithm using Bellman's equations. For each vertex $v \in \mathcal{V}$, let $\tilde{U}^\pi(v)$ be the best cost value of a path from source s to vertex v:

$$\begin{aligned} \tilde{U}^\pi(v) &= \max_{a \in \Gamma^-(v)} \left\{ \tilde{U}^\pi(\mathcal{T}(a)) + (p_a - \sum_{r=1}^{R} \pi_c c_a^r) \right\} \\ \tilde{U}^\pi(s) &= 0 \end{aligned}$$

The $\tilde{U}^\pi(v)$ values are the so-called forward labels. Note that in the case of directed acyclic graph, sets $\mathcal{T}(a)$ and $\mathcal{H}(a)$ contain only one vertex. Symmetrically a backward labelling algorithm can be implemented to compute $\tilde{C}^\pi(v)$ that denotes the reverse longest path from sink t to vertex v:

$$\begin{aligned} \tilde{C}^\pi(v) &= \max_{a \in \Gamma^+(v)} \left\{ \tilde{C}^\pi(\mathcal{H}(a)) + (p_a - \sum_{r=1}^{R} \pi_r c_a^r) \right\} \\ \tilde{C}^\pi(t) &= 0 \end{aligned}$$

Using the above longest path values, the cost of the best path which contains arc a for any given arc $a \in \mathcal{A}$ can be evaluated as follows:

$$\tilde{F}^\pi(a) = \tilde{U}^\pi(\mathcal{T}(a)) + (p_a - \sum_{r=1}^{R} \pi_r c_a^r) + \tilde{C}^\pi(\mathcal{H}(a)) + \sum_{r=1}^{R} \pi_r \mathcal{R}^r$$

Now assume a given lower bound value LB_{RCLPP} on the RCLPP problem. Then for each arc $a \in \mathcal{A}$ which does not take part in any optimal solution, one can try to filter it out. If the condition $\tilde{F}^\pi(a) < LB_{RCLPP}$ holds, arc a can be removed from the network. Equivalently, its associated variable x_a can be set to zero and this no matter the value of multipliers π. Indeed, if the

previous condition holds, this implies that the best value of a longest path which contains a is worst than a known incumbent solution associated to our lower bound. An illustration is provided in Figure 2.9. At a given iteration of some procedure to obtain valid Lagrangian multiplier vector π, the forward labelling procedure is first executes to obtain $\tilde{U}^{\pi}(v), \forall v \in \mathcal{V}$ and then follows by the backward labelling procedure to compute $\tilde{C}^{\pi}(v), \forall v \in \mathcal{V}$. Then for each arc $a \in \mathcal{A}$, the value $\tilde{F}^{\pi}(a)$ is computed and compared with LB_{RCLPP}. Arc $a \in \mathcal{A}$ such that $\tilde{F}^{\pi}(a) < LB_{RCLPP}$ are then removed from graph G.

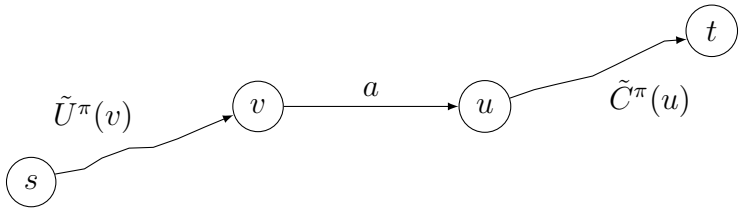

Figure 2.9: Filtering representation on graph for a given Lagrangian multiplier vector π. If the best value of a path which contains a is lower than a best known lower bound (*i.e.* if $\tilde{U}^{\pi}(v) + (p_a - \sum_{r=1}^{R} \pi_r c_a^r) + \tilde{C}^{\pi}(u) + \sum_{r=1}^{R} \pi_r \mathcal{R}^r < LB_{RCLPP}$ holds), arc a can be removed from the graph

2.3.2 Extension to the case of a hypergraph

The generic network flow problem formulation can be expressed in term of binary variables in directed acyclic graph. In a hypergraph however the flow variables are integer and one also has to consider the way flows are recombined in a hyperarc. This leads to a different mode of computation of the C values.

Consider the hypergraph flow model (2.22)-(2.26). Applying a Lagrangian relaxation on constraints (2.26) with multipliers π leads to the Lagrangian subproblem:

$$L(\pi) = \max \left\{ \sum_{i \in \mathcal{I}} (e_i - \pi_i) \sum_{a \in \mathcal{A}^0(i)} n_a(i) x_a + \sum_{i \in \mathcal{I}} \pi_i d_i \text{ s.t. } (2.23)\text{-}(2.25) \right\}$$

Just as in the case of the longest path in a graph, the computation of the Lagrangian bound on hypergraphs can be performed by a forward dynamic program starting from the sources to the unique sink. For a given $v \in \mathcal{V}^0$, $U^{\pi}(v)$ is the best value flow value from sources to vertex v:

$$U^{\pi}(v) = \max_{a \in \Gamma^-(v)} \left\{ \sum_{v' \in \mathcal{T}(a)} n_a(v') U^{\pi}(v') \right\}$$

$$U^\pi(i) = e_i - \pi_i, \forall i \in \mathcal{I}$$

This forward dynamic program is initialized with values $U^\pi(i) = e_i - \pi_i, \forall i \in \mathcal{I}$. Note that each hyperarc has no cost as the cost carries only on the boundary states.

Deriving C^π values is more complex in the hypergraph case. Let $C^\pi(v)$ be the maximum cost of a flow when $v \in \mathcal{V}^0$ is a hypergraph source. $C^\pi(v)$ is an evaluation of the remaining cost to the sink t and defined as follows:

$$C^\pi(v) = \max_{a \in \Gamma^+(v)} \left\{ C^\pi(\mathcal{H}(a)) + \sum_{v' \in \mathcal{T}(a)} n_a(v') U^\pi(v') - U^\pi(v) \right\}$$

Since $\mathcal{T}(a)$ is a multiset, it is mandatory to sum up all $U^\pi(v'), v' \in \mathcal{T}(a)$ and then to subtract $U^\pi(v)$. This is implied by the fact that v can occur more than once in $\mathcal{T}(a)$. The standard computation of $C^\pi(v)$ values is performed by a backward dynamic program once the forward recursion on U^π has been performed.

Once U^π and C^π values are obtained, filtering starts. Contrary to directed acyclic graphs, evaluation of the solution cost is done under the assumption that a hyperarc in this solution carries a flow of value at least equal to one. This difference comes from the fact that hyperarc variables are integer in a hypergraph and not binary as in a simple graph. Let $F^\pi(a)$ be the maximum cost of a flow solution containing hyperarc $a \in \mathcal{A}^0$ carrying a flow of value at least one:

$$F^\pi(a) = \sum_{v \in \mathcal{T}(a)} n_a(v) U^\pi(v) + C^\pi(\mathcal{H}(a)) + \sum_{i \in \mathcal{I}} \pi_i d_i$$

Assume now a valid lower bound value LB for the C-2KP-RE-4-r. If $F^\pi(a) < LB$ holds then hyperarc a cannot take part in any solution of the problem that is better than the incumbent one associated to LB. Consequently x_a variable can be fixed to zero or equivalently hyperarc a can be removed from the hypergraph. An illustration is given in Figure 2.10. Note that the impact of filtering depends on the quality of the lower bound value LB, and the quality of the Lagrangian multipliers.

2.3.3 Optimizing Lagrangian multipliers

The quality of Lagrangian multiplier vector π is important to have a good quality filtering. To adjust them a subgradient algorithm or a column generation or a column-and-row generation can be used.

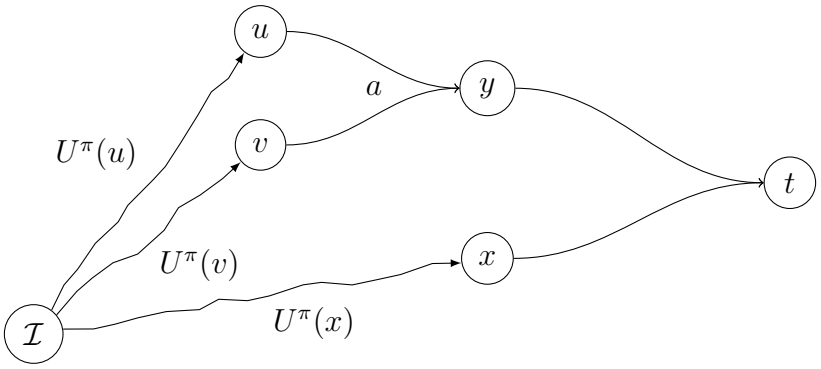

Figure 2.10: Filtering representation on hypergraph. If the best value of a flow which contains at least one time a is lower than a best known lower bound (*i.e.* if $U^\pi(u) + U^\pi(v) + C^\pi(y) + \sum_{i \in \mathcal{I}} \pi_i d_i < LB$ holds), hyperarc a can be removed from the hypergraph.

2.3.3.1 Using a subgradient algorithm

The proposed subgradient algorithm is the standard one proposed by Held et al. [37]. Starting from model (2.22)-(2.26), a Lagrangian relaxation is applied on complicated constraints (2.26). Lagrangian multipliers related to (2.26) are then obtained by computing a subgradient.

At each iteration j, for a given multiplier vector π^j, the Lagrangian bound $L(\pi^j)$ is computed by solving the forward unbounded dynamic program in which each item profit value e_i is replaced with $e_i - \pi_i^j$. Vector π is updated using:

$$\pi^{j+1} = \max\left\{0, \pi^j + \alpha \frac{(L(\pi^j) - LB)}{\|g^j\|^2} g^j\right\}$$

where α is a fixed parameter in $]0, 2]$, LB is an incumbent solution value, while g^j is a subgradient. Specifically, g_o^j represents the violation of (2.26) constraint for item of index $o = \{1, \ldots, m\}$:

$$g_o^j = \sum_{a \in \mathcal{A}^0(\mathcal{I}(o))} n_a(\mathcal{I}(o)) x_a^j - d_{\mathcal{I}(o)}$$

The subgradient procedure is stopped either after a finite number of iterations or when the best problem dual bound (*i.e.* $\min_j \{L(\pi^j)\}$) has not been improved for a parametrized number of iterations. Before each π updates, filtering occurs on G^0.

2.3.3.2 Using column generation

A way to optimize Lagrangian multipliers π is to apply column generation to the Dantzig-Wolfe reformulation of the ILP model (2.23)-(2.26). The master program assumes a set $\mathcal{J}$ of cutting patterns which can be applied on the initial stock sheet. Each cutting pattern $j \in \mathcal{J}$ is defined by its cost c_j and the number a_{ij} of items $i \in \mathcal{I}$ cut into it. The cost of pattern j is simply the sum of the profits of the cut items $c_j = \sum_{i \in \mathcal{I}} e_i a_{ij}$. Let binary variable $y_j = 1$ if pattern j is used, 0 otherwise. Using these definitions, the problem is rewritten as follows:

$$\max \sum_{j \in \mathcal{J}} c_j y_j \tag{2.32}$$

$$\text{s.t.} \sum_{j \in \mathcal{J}} y_j = 1 \tag{2.33}$$

$$\sum_{j \in \mathcal{J}} a_{ij} y_j \leqslant d_i, \quad \forall i \in \mathcal{I} \tag{2.34}$$

$$y_j \in \{0, 1\}, \quad \forall j \in \mathcal{J} \tag{2.35}$$

The objective function (2.32) is to find the combination of patterns of highest cost. Constraint (2.33) ensures that only one pattern is selected and constraint set (2.34) requires that the selected pattern does not imply item overproduction. Note that the pattern definition allows item overproduction since a pattern is obtained by solving the unbounded dynamic program related to the U-2KP-RE-4-r.

As the size of $\mathcal{J}$ is exponential, it is not practical to enumerate all patterns $j \in \mathcal{J}$. Therefore, a delayed column generation is used to solve the linear relaxation of model (2.32)-(2.35) which, in this approach, defines the master program (denoted MP_{cg}). A restricted master problem, denoted RMP_{cg}, is defined by a subset of patterns $\bar{\mathcal{J}} \subset \mathcal{J}$. To identify if a new pattern j should be added into $\bar{\mathcal{J}}$ in the hope of improving the objective value, one solves a so-called pricing problem. Its objective is the so-called reduced cost of a pattern $r_j = c_j - \sum_{i \in \mathcal{I}} a_{ij} \pi_i$ where π_i are duals associated to constraint set (2.34) in the solution to RMP_{cg}. This pricing problem is solved using the forward unbounded dynamic program in which each item profit value e_i is replaced with $e_i - \pi_i$. The method stops when the RMP_{cg} is solved to optimality. The filtering occurs after each forward dynamic program solving.

2.3.3.3 Using row-and-column generation

The column generation procedure can show slow convergence, a drawback which addressed by using row-and-column instead. The latter approach can

accelerate convergence thanks to better recombination of previously generated pricing problem solutions as outlined by Sadykov and Vanderbeck [71].

The method is applied to the LP relaxation of the ILP model given by (2.22)-(2.26), where (2.25) are replaced by setting $x_a \in \mathbb{R}, \forall a \in \mathcal{A}^0$. Let LP_{cg} be this linear program. At each iteration, LP_{cg} is solved with a restricted number of variables and constraints. The optimal dual values associated with constraints (2.26) are then used to obtain a positive reduced cost pattern (still using the forward dynamic program) as in the standard column generation approach of the previous section. Then, the flow associated to the pattern is decomposed into its hyperarcs, and the latter are added to the restricted formulation, if absent. Missing flow conservation constraints (2.23), in which the added variables participate, are also added to the restricted formulation. The validity of this algorithm follows from the fact that there exists a positive reduced cost variable x_a if and only if there is a positive reduced cost solution to the pricing problem (as proved in Sadykov and Vanderbeck [71]). Filtering is then executed using dual values π associated with constraints (2.26). The method stops when no more variables and constraints can be added to LP_{cg}.

2.4 A label setting algorithm for the bounded 2KP

Even after filtering, the size of the ILP model (2.22)-(2.26) is typically too large to be solved efficiently by a general purpose MIP solver. The alternative approach considered here is to adapt the dynamic programming solver to account for bounds on item. This is done by extending the state space by including the current usage of each item. It results in an exponential growth of the state space, which makes impractical a direct Bellman's algorithm. However specific techniques have emerged in the the last decade to tackle such large size dynamic programs, specifically in the literature on the Elementary Resource Constrained Shortest Path Problem (ERCSPP). The most efficient ones combined so-called *label setting* algorithms and *Decremental State Space Relaxation* (DSSR) (see *e.g.* Righini and Salani [68] and Martinelli et al. [57]). These methods iteratively consider a sequence of dynamic programs related to relaxations of some resource constraints. The state space is then enriched by adding a currently violated resource constraint in the state space until a feasible solution is found. A possible enhancement consists in using a variable fixing procedure (or filtering procedure) to speed up solution time (see Irnich et al. [44] and Detienne et al. [22]). Although these methods proved their strength on graphs, applying them to large hypergraphs is not straightforward.

The purpose of this section is to describe a generic procedure to solve hypergraph flow problem with side-constraints. First, a description is done on how the unbounded dynamic program can be extended to take into account the

item upper bounds resulting in an extended dynamic program. This extended dynamic program is then solved by a forward labelling algorithm. Finally an exact algorithm combining the forward labelling algorithm with DSSR strategy together with Lagrangian cost filtering to solve the C-2KP-RE-4-r problem to optimality is detailed.

2.4.1 Dynamic program for the bounded case

As outlined by Christofides and Whitlock [10] and Velasco and Uchoa [85], it is possible to write a dynamic program which handles item upper bounds. Consequently with this new dynamic program, one needs to perform a forward search to obtain the optimal solution of the associated bounded 2KP. Hereinafter, the dynamic program (2.1)-(2.6) is extended in the same fashion.

Given $m = |\mathcal{I}|$, each defined state $(w,h)^j$ and $\overline{(w,h)^j}$ in the unbounded dynamic program (2.1)-(2.6) is enriched with a vector $\mathbf{Q} \in \mathbb{N}^m$. This vector models the number of items which are cut in the residual cutting problem associated with the considered state. This leads to define new states $(w,h,\mathbf{Q})^j$ and $\overline{(w,h,\mathbf{Q})^j}$. Each of them corresponds to a plate of dimension $w \times h$ at cutting stage j where items are cut exactly Q_i times. Thus $U((w,h,\mathbf{Q})^j)$ and $U(\overline{(w,h,\mathbf{Q})^j})$ are the maximum values related to states $(w,h,\mathbf{Q})^j$ and $\overline{(w,h,\mathbf{Q})^j}$. The initial bound vector is defined by $\mathbf{D} = (d_1,\ldots,d_m)$. To ease the presentation, let consider notations $\mathbf{Q}' - \mathbf{Q}''$ and $\mathbf{Q}' + \mathbf{Q}''$ to indicate the component-wise difference and sum, and $\mathbf{Q}' \leqslant \mathbf{Q}''$ to indicate that $Q'_i \leqslant Q''_i, \forall i \in \mathcal{I}$. The bold-face notation $\mathbf{i}$ indicates the vector in $\{0,1\}^n$ with component i equal to 1 and the others to 0. Notation $\mathbf{0}$ refers to the vector with all components equal to zero. The extended recursion takes the form:

$$U((w,h,\mathbf{Q})^1) = \max\left\{0, \max_{\substack{i\in\bar{\mathcal{I}}:h_i\leqslant h,w_i\leqslant w,\\ Q'_i\geqslant 1,\mathbf{Q}'\leqslant\mathbf{Q}}} \left\{ \begin{array}{l} U(\overline{(w_i,h,\mathbf{Q}')^2})+ \\ U((w-w_i,h,\mathbf{Q}-\mathbf{Q}')^1) \end{array} \right\}\right\} \quad (2.36)$$

$$U((w,h,\mathbf{Q})^2) = \max\left\{0, \max_{\substack{i\in\bar{\mathcal{I}}:h_i\leqslant h,w_i\leqslant w,\\ Q'_i\geqslant 1,\mathbf{Q}'\leqslant\mathbf{Q}}} \left\{ \begin{array}{l} U(\overline{(w,h_i,\mathbf{Q}')^3})+ \\ U((w,h-h_i,\mathbf{Q}-\mathbf{Q}')^2) \end{array} \right\}\right\} \quad (2.37)$$

$$U((w,h,\mathbf{Q})^3) = \max\left\{0, \max_{\substack{i\in\bar{\mathcal{I}}:h_i\leqslant h,w_i\leqslant w,\\ Q'_i\geqslant 1,\mathbf{Q}'\leqslant\mathbf{Q}}} \left\{ \begin{array}{l} U(\overline{(w_i,h,\mathbf{Q}')^4})+ \\ U((w-w_i,h,\mathbf{Q}-\mathbf{Q}')^3) \end{array} \right\}\right\} \quad (2.38)$$

$$U(\overline{(w,h,\mathbf{Q})^2}) = \max_{\substack{i\in\bar{\mathcal{I}}:w_i=w,h_i\leqslant h,\\ Q_i\geqslant 1}} \left\{e_i + U((w,h-h_i,\mathbf{Q}-\mathbf{i})^2)\right\} \quad (2.39)$$

$$U(\overline{(w,h,\mathbf{Q})^3}) = \max_{\substack{i\in\bar{\mathcal{I}}:h_i=h,w_i\leqslant w,\\ Q_i\geqslant 1}} \left\{e_i + U((w-w_i,h,\mathbf{Q}-\mathbf{i})^3)\right\} \quad (2.40)$$

$$U(\overline{(w,h,\mathbf{Q})^4}) = \max\left\{0, \max_{\substack{i\in\bar{\mathcal{I}}: w_i=w, h_i\leqslant h,\\ Q_i\geqslant 1}}\left\{e_i + U(\overline{(w,h-h_i,\mathbf{Q}-\mathbf{i})^4})\right\}\right\} \quad (2.41)$$

The optimal solution to the dynamic program of the bounded 2KP is obtained by computing $\max_{\mathbf{Q}\leqslant\mathbf{D}}\{U((W,H,\mathbf{Q})^1)\}$. Note that all possible $\mathbf{Q}'\leqslant\mathbf{Q}$ have to be considered in recurrence relations (2.36)-(2.41). This contributes to increase the state space size. This extended dynamic program size is multiplied by $\prod_{i\in\mathcal{I}}(d_i+1)$ in comparison with the its unbounded variant. Clearly this extended dynamic program is not computational when the number of items is large.

Using the paradigm of Martin et al. [56] and as shown in Section 2.1.2, the hypergrah representation of this extended dynamic program entails an extended hypergraph. The used notation for this hypergraph is $G^m = (\mathcal{V}^m, \mathcal{A}^m)$, where superscript m represents the dimension of vector $\mathbf{Q}$. Note that hypergraph $G^0 = (\mathcal{V}^0, \mathcal{A}^0)$ representing the unbounded dynamic program is a projection of hypergraph G^n. This projection maps each vertex $v^m \in \mathcal{V}^m$, which corresponds to a state $(w,h,\mathbf{Q})^j$ or to a state $\overline{(w,h,\mathbf{Q})^j}$, to a vertex $v^0 \in \mathcal{V}^0$, which corresponding to state $(w,h)^j$ or $\overline{(w,h)^j}$ respectively. Such projection is obtained by simply dropping the vector $\mathbf{Q}$ of a state $(w,h,\mathbf{Q})^j$ and consequently this state reduces itself to $(w,h)^j$. Note that this projection is surjective since two states $(w,h,\mathbf{Q})^j$ and $(w,h,\mathbf{Q}')^j$ projects into the same state $(w,h)^j$. Thus every vertex $v^m \in \mathcal{V}^m$ can be denoted as $(v^0,\mathbf{Q})$, where v^0 is its projection into the state space of (2.1)-(2.6). In the same way hyperarc $a^0 \in \mathcal{A}^0$ is the projection of hyperarc $a^m \in \mathcal{A}^m$ if $\mathcal{H}(a^0)$ is the projection of vertex $\mathcal{H}(a^m)$, while each vertex in $\mathcal{T}(a^m)$ has its respective projection on a vertex in $\mathcal{T}(a^0)$.

2.4.2 Forward labelling in the extended space

The bounded dynamic program is intractable due to its size when a large number of items are considered. Nevertheless a way to solve it is to use a forward labelling algorithm. This type of algorithm is a dynamic program implementation in which states are created recursively as labels starting from an empty solution. The focus is to create only states that correspond to feasible partial solutions.

A feasible partial solution is defined by a label L that takes the form of a tuple $(p_L, v_L^0, \mathbf{Q}_L)$. p_L denotes its profit computed using $\mathbf{Q}_L$, $v_L^0 \in \mathcal{V}^0$ is the plate status, and $\mathbf{Q}_L$ is the item status related to the partial solution L. One can store labels in a bucket based on their corresponding plate v_L^0. For a given vertex $v^0 \in \mathcal{V}^0$, let $\mathcal{L}(v^0)$ be the set of labels L such that $v_L^0 = v^0$. For each label L, there exists a vertex $v^m \in \mathcal{V}^m$ such that $v^m = (v_L^0, \mathbf{Q}_L)$. Note that $\mathbf{Q}_L$ is one of the possible $\mathbf{Q}'\leqslant\mathbf{Q}$ considered in (2.36)-(2.41).

To avoid complete exploration of all labels (*i.e.* partial solutions), one can avoid some of them by application of a dominance principle. The purpose of such dominance is to establish that a partial solution can be discarded without optimality loss. Consider two labels L and L' taken from $\mathcal{L}(v^0), v^0 \in \mathcal{V}^0$. Formally, label L dominates label L' (*i.e.* $L \succeq L'$) if one can guarantee that any extension of L' cannot be strictly better than the best extension of L. One can define a weak dominance check $L \succeq_{weak} L'$ related to verify that $p_L \geqslant p_{L'}, v^0_L = v^0_{L'}, \mathbf{Q}_L = \mathbf{Q}_{L'}$. One can also define a strong dominance check $L \succeq L'$ related to verify that $p_L \geqslant p_{L'}, v^0_L = v^0_{L'}, \mathbf{Q}_L \leqslant \mathbf{Q}_{L'}$. Observe that both rules are to be applied within a label bucket for a fixed vertex v^0. In other words, a label may dominate another one only if both labels are in the same bucket. While strong dominance is applied only in some algorithms, the weak dominance check is maintained at all time. For a given hypergraph vertex $v^m = (v^0, \mathbf{Q})$, a unique label $L = L(v^m)$, which is the one of largest profit p_L amongst all those with $(v^0_L, \mathbf{Q}_L) = (v^0, \mathbf{Q})$. Note that, since $p_L = \sum_{i \in \mathcal{I}} e_i Q_{L,i}$, weak and strong dominances reduce to $p_L = p_{L'}, v^0_L = v^0_{L'}, \mathbf{Q}_L = \mathbf{Q}_{L'}$. The reason why such distinction is made here is that below the state-space relaxation method is introduced, in which vector $\mathbf{Q}_L$ has a dimension lower than m. In that case, $p_L \neq \sum_{i \in \mathcal{I}} e_i Q_{L,i}$, and the strong dominance may hold between a pair of labels when the weak dominance does not.

Instead of explicitly generating hypergraph G^m, the forward recursion is implemented in the projected hypergraph G^0. The recursion is initialized by defining the sources of the extended hypergraph. This correspond to create item labels $(e_i, v^0, \mathbf{i})$ and waste label $(0, v^0, \mathbf{0})$ which are then put in the bucket associated to the original vertex $v^0 \in \mathcal{V}^0$. Then vertices $v^0 \in \mathcal{V}^0$ are considered in topological order. For each vertex $v^0 \in \mathcal{V}^0$, labels related to this vertex are built using all hyperarcs $a^0 \in \Gamma^-(v^0)$ and the previously built labels for each tail in $\mathcal{T}(a^0)$. Observe, indeed, that selecting a^0 induces a recombination of labels associated with each of the tail vertices of a^0. This is formalized below.

Given hyperarc $a^0 \in \Gamma^-(v^0)$, assume that $\mathcal{T}(a^0)$ takes the explicit form $\{v^0_1, v^0_2, \ldots, v^0_f\}$ where the same vertex may occur several times. Let $\mathcal{E}^m(a^0)$ be the set of possible transitions defined as recombinations of partial solutions that are induced by a^0:

$$\mathcal{E}^m(a^0) = \mathcal{L}(v^0_1) \times \mathcal{L}(v^0_2) \times \ldots \times \mathcal{L}(v^0_f) \tag{2.42}$$

An element $E^m(a^0) \in \mathcal{E}^m(a^0)$ is called a transition. It consists in selecting a label for each tail vertex in $\{v^0_1, v^0_2, \ldots, v^0_f\}$ using related buckets $\{\mathcal{L}(v^0_1), \mathcal{L}(v^0_2), \ldots, \mathcal{L}(v^0_f)\}$. Observe that the associated tail multiset $\mathcal{T}(a^0)$ may contain more than one time the same vertex (*i.e.* there exists f' and f'' such that $v^0_{f'} = v^0_{f''}$). However this does not imply that the selected labels $L(v^0_{f'})$ and $L(v^0_{f''})$ in buckets $\mathcal{L}(v^0_{f'})$ and $\mathcal{L}(v^0_{f''})$ for the f'-th and f''-th vertices in $\mathcal{T}(a^0)$ are necessarily the same.

A transition $E^m(a^0) \in \mathcal{E}^m(a^0)$ (or using simpler notation $E \in \mathcal{E}^m(a^0)$) defines a combination of labels $L' \in E$ used to create a new label L of the form:

$$(p_L, v_L^0, \mathbf{Q}_L) = \left(\sum_{L' \in E} p_{L'}, \mathcal{H}(a^0), \sum_{L' \in E} \mathbf{Q}_{L'} \right) \tag{2.43}$$

A transition E is said to be valid if the created label L ensures that item bound constraints are valid (*i.e.* $\mathbf{Q}_L \leqslant \mathbf{D}$). The obtained label L from transition E defines a hyperarc $a^m \in \mathcal{A}^m$ of the form:

$$\mathcal{H}(a^m) = \left\{(v_L^0, \mathbf{Q}_L)\right\} \text{ and } \mathcal{T}(a^m) = \left\{(v_{L'}^0, \mathbf{Q}_{L'})\right\}_{L' \in E} \tag{2.44}$$

From the definition of a^m, observe that $\mathcal{H}(a^m)$ defines the vertex $v^m = (v_L^0, \mathbf{Q}_L), v^m \in \mathcal{V}^m$. Since a valid transition E implies to create a hyperarc a^m, let $E(a^m)$ be the set of labels in transition E defining the tail vertices of a^m. One can also observe that there is a mapping between hyperarcs a^m and a^0. For further reference, this hyperarc-mapping $\mathcal{M}^m(a^0) \subset A^m$ is the set of hyperarcs a^m that project onto a^0:

$$\mathcal{M}^m(a^0) = \left\{a^m \in A^m : \{v_{L'}^0\}_{L' \in E(a^m)} = \mathcal{T}(a^0) \text{ and } \{v_{L(\mathcal{H}(a^m))}^0\} = \mathcal{H}(a^0)\right\} \tag{2.45}$$

From previous notations, a pseudo-code of the forward labelling algorithm is given in Algorithm 3. The algorithm considers all vertices $v^0 \in \mathcal{V}^0$ in topological order. If the vertex v^0 is a boundary states it is initialized. Otherwise new labels are created using predecessor set $\Gamma^-(v^0)$. For a given predecessor $a^0 \in \Gamma^-(v^0)$, the set of transitions $\mathcal{E}^m(a^0)$ is computed. For each valid transition E, the associated label L is created and then stored in bucket $\mathcal{L}(v^0)$. Finally some labels are removed from $\mathcal{L}(v^0)$ using the weak dominance check. At the end of the algorithm, the label $L \in \mathcal{L}(t)$ of maximum value (*i.e.* the optimal solution) with t the sink of G^0 is returned. This labelling algorithm can be used in practice when the size of $\mathbf{Q}$ is small. Since the size of $\mathbf{Q}$ is related to the number of items in a 2KP instance, it does not seem practical at all to solve directly the problem using Algorithm 3. Indeed difficulties will be encountered due to huge state space size and computation time. To avoid this drawback, the forward labelling algorithm is embedded in the Decremental

State Space Relaxation.

Algorithm 3: Forward Labelling Algorithm for Hypergraph G^m

for $v^0 \in \mathcal{V}^0$ *in topological order* **do**
  if $\Gamma^-(v^0) = \varnothing$ **then**
    if v^0 *corresponds to an item* $i \in \mathcal{I}$ **then** $\mathcal{L}(v^0) \leftarrow \{(e_i, v^0, \mathbf{i})\}$
    else $\mathcal{L}(v^0) \leftarrow \{(0, v^0, \mathbf{0})\}$
  else
    for $a^0 \in \Gamma^-(v^0)$ **do**
      compute $\mathcal{E}^m(a^0)$
      for $E \in \mathcal{E}^m(a^0)$ **do**
        $L \leftarrow (\sum_{L' \in E} p_{L'}, v^0, \sum_{L' \in E} \mathbf{Q}_{L'})$
        if $\mathbf{Q}_L \leqslant \mathbf{D}$ **then** $\mathcal{L}(v^0) \leftarrow \mathcal{L}(v^0) \bigcup \{L\}$
    apply weak dominance check on $\mathcal{L}(v^0)$
return $\max_{L \in \mathcal{L}(t)} p_L$ for t being the sink of G^0 and optimal solution S^*.

2.4.3 Decremental State Space Relaxation

Running the above forward labelling algorithm becomes quickly impractical even on medium size instances, given the huge size of the extended state space. The strongly exponential growth in the state space is induced by the size of $\mathbf{Q}$ that is used to keep track of the number of items that have been cut.

The main motivation to avoid to work in the extended state space $\mathcal{S}^m$ given by (2.36)-(2.41) is that in a valid solution only some items in $\mathcal{I}$ are attractive and will be in an optimal solution. Hence an active strategy can be used to identify those attractive items and only keep track of them. Idea is to work with an item usage vector $\mathbf{Q}^{m'} \in \mathbb{N}^{m'}$ with $m' < m$, related to considering only a subset $\mathcal{I}^{m'} \subset \mathcal{I}$. Main advantage is that working with $\mathbf{Q}^{m'}$ leads to a state space size smaller than the one related to use $\mathbf{Q}^m$. Drawback is that the problem to solve is a relaxation of the initial one since only a subset $\mathcal{I}^{m'} \subset \mathcal{I}$ is considered. The generic strategy that underlies this state-space reduction method is known as *Decremental State Space Relaxation* (DSSR). Such approach has proved to be efficient on C-2KP-NR-k-f (see Christofides and Hadjiconstantinou [9] and Velasco and Uchoa [85]) and on Vehicle Routing Problems (see Righini and Salani [68] and Martinelli et al. [57]) among others.

Formally the technique works as follows. In the extended state space $\mathcal{S}^m$ defined by equations (2.36)-(2.41), the state associated to a label is a pair $(v, \mathbf{Q}^m)$ of dimension $m + 1$ (m for the item usage vector and 1 for the vertex identification). In a projected state space $\mathcal{S}^{m'}$, with $m' < n'$, a state is defined by a pair $(v, \mathbf{Q}^{m'})$ of dimension $m' + 1$ (m' for the item usage vector and 1 for the vertex identification). Then, the number of states to explore is reduced. The size of state space $\mathcal{S}^m$ is $O\left(|\mathcal{V}^0| \cdot Pi_{i=1}^{m}(d_i + 1)\right)$. For its projection in

dimension $m' + 1$, the size of the state space becomes $O\left(|\mathcal{V}^0| \cdot \Pi_{i \in \mathcal{I}^{m'}}(d_i + 1)\right)$ with $|\mathcal{I}^{m'}| = m'$.

Working in the projected state space $\mathcal{S}^m$ amounts to considering a relaxation of the bounded 2KP problem, as one cannot guarantee the feasibility of the solution regarding demand constraints (2.26). Some partial solutions associated to a projected state $(v, \mathbf{Q}^{m'})$ of $\mathcal{S}^{m'}$ can yield an item production higher than the demand for items $i \in \mathcal{I} \backslash \mathcal{I}^{m'}$. Although the optimal solution of the relaxation on $\mathcal{S}^{m'}$ may not be feasible, it provides a valid dual bound. But, interestingly, if the optimal solution in state space $\mathcal{S}^{m'}$ is feasible in state space $\mathcal{S}^m$, then it is also optimal in $\mathcal{S}^m$. Figure 2.11 pictures a mapping of states $s^m \in \mathcal{S}^m$ into states $s^{m'} \in \mathcal{S}^{m'}$, some are feasible, others are infeasible. Observe that definitions of extensions (2.43), extended hyperarcs (2.44), and mappings (2.45) can easily be recasted for any m' with $0 \leqslant m' \leqslant m$, so are the associated definitions of $\mathcal{E}^m(a^0)$. Hence, Algorithm 3 can be used to solve the relaxed problem for a given state space relaxation $\mathcal{S}^{m'}$, simply by replacing m by m'.

A natural dynamic strategy to update the state space derives from the above discussion. If the optimal solution to the problem related to a smaller state space $\mathcal{S}^{m'}$ is feasible for the original problem this solution is also optimum and problem optimality is reached. Otherwise, when the best solution in $\mathcal{S}^{m'}$ is not feasible in $\mathcal{S}^m$, the state space $\mathcal{S}^{m'}$ is expanded. Indeed having the best solution in $\mathcal{S}^{m'}$ not feasible in $\mathcal{S}^m$ implies that at least one item $i \in \mathcal{I} \backslash \mathcal{I}^{m'}$ is cut more than d_i times. The state space expansion is then dictated by violated demand constraints (2.26). An extra dimension is added to $\mathcal{S}^{m'}$ by considering $\mathcal{I}^{m'+1} = \mathcal{I}^{m'} \cup \{i\}$. Such dynamic state space expansion is a well-known technique used on scheduling problem (see Ibaraki and Nakamura [43]).

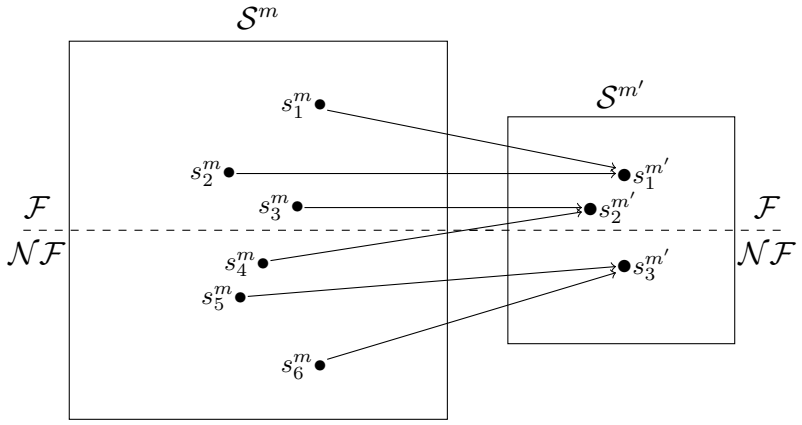

Figure 2.11: Example of the projection of feasible ($\mathcal{F}$) and non feasible ($\mathcal{NF}$) states in $\mathcal{S}^m$ to $\mathcal{S}^{m'}$. In this example, non feasible state s_4^m becomes a feasible state in $\mathcal{S}^{m'}$.

The proposed approach starts with $\mathcal{I}^{m'} = \varnothing, m' = 0$. The associated map-

ping function projects extended state space given by (2.36)-(2.41) into state space of the unbounded dynamic program given by (2.1)-(2.6). At iteration m', a new set $\mathcal{I}^{m'+1}$ is obtained by adding one overproduced item to $\mathcal{I}^{m'}$. The process is repeated until the solution produced at iteration m' admits a feasible expansion in $\mathcal{S}^m$. The method converges toward the optimal value after at most $m+1$ iterations leading to solve the extended dynamic program (2.36)-(2.41). The dynamic program used at iteration m' can be represented by a hypergraph $G^{m'} = (\mathcal{V}^{m'}, \mathcal{A}^{m'})$ in which every vertex is of the form $v^{m'} = (v^0, \mathbf{Q}^{m'}) \in \mathcal{V}^{m'}$. A possible implementation for vector $\mathbf{Q}^{m'}$ is to consider this vector as having m dimensions, but fixing $Q_i^m = 0$ for all $i \notin \mathcal{I}^{m'}$, while keeping track of an index set $\mathcal{I}^{m'}$ associated to the current iteration m'. Although such procedure could rely on applying Algorithm 3 with m replaced by m', one can do better than restart from scratch at each iteration. Moreover, one can take advantage of filtering methods to reduce the hypergraph size at each iteration.

2.4.4 Iterative labelling algorithm

In the DSSR strategy context, it is better to avoid hypergraph creation from scratch at iteration $m', 1 < m' \leqslant m$. Nevertheless, one can use hypergraph of iteration $(m'-1)$ to create the one at iteration m'. The idea is to warm start creation of hypergraph $G^{m'}$ at iteration m' with input hypergraph $G^{m'-1}$ from iteration $(m'-1)$. Note that the projection relation between hypergraphs $G^{m'-1}$ and $G^{m'}$ is simply defined by associating a vertex $v^{m'} = (v^0, \mathbf{Q}^{m'})$ to its projection $v^{m'-1} = (v^0, \mathbf{Q}^{m'-1})$ by setting $Q_i^{m'-1} = Q_i^{m'}$ for all $i \in \mathcal{I}^{m'-1}$ and $Q_i^{m'-1} = 0$ otherwise. Similarly, a hyperarc $a^{m'}$ can be projected on the hyperarc $a^{m'-1}$ that is defined by its projected tails and head. Moreover, as a reminder, each vertex $v^{m'}$ (resp. $v^{m'-1}$) is associated to a unique label $L^{m'}$ (resp. $L^{m'-1}$) given that weak dominance is maintained at all time, recording only the partial solution of best profit value.

The main motivation for warm starting is to take advantage of filtering done up to iteration $(m-1)$ to limit the hypergraph building effort at iteration m. However, there are two important remarks to be aware of in such incremental scheme:

1. The dominance rule is valid only within the current iteration: *i.e.* labels that are dominated at iteration $(m'-1)$ can become non-dominated at iteration m', once a new item is recorded in the vector $\mathbf{Q}$. Hence, dominated labels L from iteration $(m'-1)$ have to be kept in order to build extensions at iteration m. In practice, instead of keeping track of a dominated label L from iteration $(m'-1)$, all hyperarcs which allow to reach label L from iteration $(m'-1)$ are stored, having both dominated and non-dominated labels as heads, but only non-dominated labels as tails. These hyperarcs are recorded in containers $\mathcal{M}^{m'-1}(a^0)$ if they projected to $a^0 \in \mathcal{A}^0$. All of them are considered in building extensions.

2. Elimination of labels by the strong dominance check is not compatible with Lagrangian filtering when Lagrangian multipliers are used (*i.e.* when $\pi \neq 0$). Indeed, dominance is evaluated based on the true profit value with $\pi = 0$. While in Lagrangian filtering, the profit becomes the reduced cost for Lagrangian multipliers π up to the tails and beyond the head node. This evaluation became wrong if intermediate vertices have been eliminated through dominance using the true cost measure. Hence, when the strong dominance check is used (*i.e.* when the parameter $enforceStrongDominance$ is true), one can only apply plain filtering for $\pi = 0$ to ensure compatibility between cost measures. Thus, two implementation strategies are considered whether parameter $enforceStrongDominance$ is true or not.

This being said observe that filtering done in iteration $(m' - 1)$, for a fixed set of Lagrangian multipliers π, remains valid in iteration m' and further iterations for the same π. Consider that Lagrangian cost filtering is applied to $G^{m'}, m' > 0$ in the same way as for G^0. If a hyperarc $a^{m'-1} \in \mathcal{A}^{m'-1}$ is filtered out, then any hyperarc $a^r \in \mathcal{A}^r$ for $r \geqslant m'$, that projects onto $a^{m'-1}$ can be filtered out too. First, observe that if $v^{m'-1} \in \mathcal{V}^{m'-1}$ is the projection at iteration $(m' - 1)$ of $v^r \in \mathcal{V}^r$, for any iteration $r, m' \leqslant r \leqslant m$, then $U^\pi(v^{m'-1}) \geqslant U^\pi(v^r)$ because iteration $(m' - 1)$ defines a relaxation of iteration r. The equation defining $C^\pi(v^{m'-1})$ can be rewritten as:

$$C^\pi(v^{m'-1}) = \max_{a^{m'-1} \in \Gamma^+(v^{m'-1})} \left\{ C^\pi(\mathcal{H}(a^{m'-1})) + \sum_{v' \in \mathcal{T}(a^{m'-1})} n_{a^{m'-1}}(v') U^\pi(v') - U^\pi(v) \right\}$$

In the same way as $U^\pi(v^{m'-1}) \geqslant U^\pi(v^r)$, it follows that $C^\pi(v^{m'-1}) \geqslant C^\pi(v^r)$, and $F^\pi(a^{m'-1}) \geqslant F^\pi(a^r)$ which explains why any filtering at iteration $m' - 1$ remains valid for any further iteration $r, m' \leqslant r \leqslant m$.

Hence, when building hypergraph $G^{m'}$ from hypergraph $G^{m'-1}$, preprocessing and filtering done up to iteration $(m' - 1)$ are used to speed up hypergraph $G^{m'}$ building process. Since hypergraph $G^{m'-1}$ is used to construct hypergraph $G^{m'}$ and strong dominance have to be ensured, previous notations about transitions have to be extended. When the weak dominance check is used, all labels in $\mathcal{L}^{m'}(v^0), v^0 \in \mathcal{V}^0$ cannot be dominated by another label in this bucket. This is no longer tru when the strong dominance check is applied. When strong dominance is used, one can have two labels considered as non-dominated according to the weak dominance but not regarding the strong one. This implies to redefine generator $\mathcal{E}^{m'}(a^{m'-1})$ to handle strong dominance. As a reminder, $\mathcal{L}^{m'}(v^0)$ is the set of labels at iteration m' related to vertex $v^0 \in \mathcal{V}^0$. Let $\hat{\mathcal{L}}^{m'}(v_f^{m'-1})$ be a subset of non-dominated labels extract from $\mathcal{L}^{m'}(v_f^0)$ such that $v_f^{m'-1}$ projects to v_f^0, and v_f^0 is the f-th tail of arc $a^0 \in \mathcal{M}^{m'-1}(a^0)$. The

set $\hat{\mathcal{E}}^{m'}(a^{m'-1})$ of transitions as recombinations of non-dominated labels at iteration m' is defined as follows:

$$\hat{\mathcal{E}}^{m'}(a^{m'-1}) = \hat{\mathcal{L}}^{m'}(v_1^{m'-1}) \times \hat{\mathcal{L}}^{m'}(v_2^{m'-1}) \times \ldots \times \hat{\mathcal{L}}^{m'}(v_f^{m'-1}) \tag{2.46}$$

Each non-dominated label in $\hat{\mathcal{L}}^{m'}(v_{f'}^{m'-1})$ projects on a non-dominated label associated to a vertex $v_{f'}^{m'-1}$ that is a f'-th tail of hyperarc $a^{m'-1}$ with f tails that was not filtered out.

When the strong dominance rule is used, recall that non-dominated labels which are associated to dominated labels in G^{m-1} have to be extended. Let $\bar{\mathcal{L}}^{m'}(v^0)$ be the subset of labels in $\mathcal{L}^{m'}(v^0)$ associated to non-dominated labels in $G^{m'-1}$. The set of transitions $\bar{\mathcal{E}}^{m'}(a^0)$ as combinations of labels at iteration m' is define as follows:

$$\bar{\mathcal{E}}^{m'}(a^0) = \left\{\mathcal{L}^{m'}(v_1^0) \times \ldots \times \mathcal{L}^{m'}(v_f^0)\right\} \setminus \left\{\bar{\mathcal{L}}^{m}(v_1^0) \times \ldots \times \bar{\mathcal{L}}^{m}(v_f^0)\right\} \tag{2.47}$$

Algorithm 4 details how to build iteratively hypergraphs. First, parameters are initialized and the unbounded problem is solved using dynamic programming (lines 2-4). If the optimal solution related to hypergraph G^0 is feasible regarding demand constraints (2.26), the method stops since optimal solution is found. Otherwise, some preprocessing occurs before building an extended hypergraph. A heuristic is first used to obtain a feasible solution S. Then using solution S, filtering or Lagrangian filtering starts depending on m' value. This choice is motivated by the fact that performing Lagrangian filtering on an extended hypergraph is time consuming since the hypergraph size depends on the size of $\mathcal{I}^{m'}$. Since the optimal solution S^* is not feasible, an overproduced item i in this solution is added to $\mathcal{I}^{m'}$ (lines 5-8). Since a new item to not overproduce has been added in $\mathcal{I}^{m'}$, the hypergraph $G^{m'}$ can be built. Vertices $v^0 \in \mathcal{V}^0$ are then crossed in topological order. If the vertex is a boundary state related to an item or to the waste vertex, the corresponding label is initialized (lines 10-11). Otherwise, labels are created using $\Gamma^-(v^0)$. For each $a^0 \in \Gamma^-(v^0)$, the mapping container is set to $\varnothing$. Then for each hyperarc $a^{m'-1}$ which projects into a^0 using $\mathcal{M}^{m'-1}(a^0)$, the set of transitions $\hat{\mathcal{E}}^{m'}(a^{m'-1})$ is built. For each transition $E \in \hat{\mathcal{E}}^{m'}(a^{m'-1})$, if the associated label L is valid, it is stored in bucket $\mathcal{L}^{m'}(v^0)$. The valid transition E or its hyperarc representation $a^{m'}$ is then stored in $\mathcal{M}^{m'}(a^0)$ for further expansion (lines 17-21). If the strong dominance check is applied, new labels from dominated labels at previous iteration have to be built. The set of transitions $\bar{\mathcal{E}}^{m'}(a^0)$ related to combinations of labels which are not related to a non-dominated at iteration $(m'-1)$ is built. As previously, each label L obtained by a transition $E \in \bar{\mathcal{E}}^{m'}(a^0)$ is built and its validity checked. Valid labels are stored in $\mathcal{L}^{m'}(v^0)$ and associated transition in $\mathcal{M}^{m'}(a^0)$ (lines 23-29). Once all labels are built

and stored in $\mathcal{L}^{m'}(v^0)$, dominated ones are removed from $\mathcal{L}^{m'}(v^0)$ using the strong or weak dominance check. When all vertices $v^0 \in \mathcal{V}^0$ are crossed, the label of best profit in $\mathcal{L}^{m'}(t)$ and the associated solution S^* are retrieved. If S^* is feasible regarding demand constraints (2.26), the method stops since S^* is optimal, a new hypergraph is built otherwise since it implies that at least an item is still overproduced.

Algorithm 4: Iterative Forward Labelling Algorithm

build hypergraph G^0
$m' \leftarrow 0, \mathcal{I}^{m'} \leftarrow \varnothing$, $\mathcal{M}^0(a^0) = \{a^0\} \quad \forall a^0 \in \mathcal{A}^0$
run Algorithm 3 on G^0 and record obtained optimal solution S^*
record singleton label bucket $\mathcal{L}^0(v^0) = \{(p^0, v^0, 0)\}, \forall v^0 \in G^0$
while S^* *is not feasible* **do**
    start a heuristic to find a feasible solution S
    if $m' = 0$ **then** perform Lagrangian filtering on $G^{m'}$ using S **else** perform filtering on $G^{m'}$ using S remove filtered hyperarcs $a^{m'}$ from $\mathcal{M}^{m'}(a^0), \forall a^0 \in A^0$
    $m' \leftarrow m' + 1$ and add to $\mathcal{I}^{m'}$ an item which bound is violated in S^*
    for $v^0 \in \mathcal{V}^0$ *in topological order* **do**
        if $\Gamma^-(v^0) = \varnothing$ **then**
            if v^0 *is a boundary state* $i \in \mathcal{I}^{m'}$ **then** $\mathcal{L}^{m'}(v^0) \leftarrow \{(e_i, v^0, \mathbf{i})\}$ **else if** v^0 *is a boundary state* $i \notin \mathcal{I}^{m'}$ **then** $\mathcal{L}^{m'}(v^0) \leftarrow \{(e_i, v^0, \mathbf{0})\}$ **else if** v^0 *corresponds to waste* **then** $\mathcal{L}^{m'}(v^0) \leftarrow \{(0, v^0, \mathbf{0})\}$
        else
            for $a^0 \in \Gamma^-(v^0)$ **do**
                $\mathcal{M}^{m'}(a^0) \leftarrow \varnothing$
                for $a^{m'-1} \in \mathcal{M}^{m'-1}(a^0)$ **do**
                    compute $\hat{\mathcal{E}}^{m'}(a^{m'-1})$
                    for $E \in \hat{\mathcal{E}}^{m'}(a^{m'-1})$ **do**
                        $L \leftarrow \left(\sum_{L' \in E} p_{L'}, v^0, \sum_{L' \in E} \mathbf{Q}^{m'}_{L'}\right)$
                        if $\mathbf{Q}^{m'}_L \leqslant \mathbf{D}$ **then**
                            $\mathcal{L}^{m'}(v^0) \leftarrow \mathcal{L}^{m'}(v^0) \cup \{L\}$
                            $\mathcal{M}^{m'}(a^0) \leftarrow \mathcal{M}^{m'}(a^0) \cup \{E\}$
                if $enforceStrongDominance$ **then**
                    compute $\bar{\mathcal{E}}^{m'}(a^0)$
                    for $E \in \bar{\mathcal{E}}^{m'}(a^0)$ **do**
                        $L \leftarrow \left(\sum_{L' \in E} p_{L'}, v^0, \sum_{L' \in E} \mathbf{Q}^{m'}_{L'}\right)$
                        if $\mathbf{Q}^{m'}_L \leqslant \mathbf{D}$ **then**
                            $\mathcal{L}^{m'}(v^0) \leftarrow \mathcal{L}^{m'}(v^0) \cup \{L\}$
                            $\mathcal{M}^{m'}(a^0) \leftarrow \mathcal{M}^{m'}(a^0) \cup \{E\}$
                    apply strong dominance check on $\mathcal{L}^{m'}(v^0)$
                else
                    apply weak dominance check on $\mathcal{L}^{m'}(v^0)$
    $S^* \leftarrow$ best solution in $\mathcal{L}^{m'}(t)$ with t the sink of G^0

2.5 Heuristics for the bounded 2KP

Due to the large size of instances to solve and to initialize the Lagrangian cost filtering, a natural approach is to use heuristics to obtain an initial feasible solution. Since the 2KP can be represented using a hypergraph, a heuristic based on it is first described hereinafter. In a second part, an evolutionary algorithm is detailed.

2.5.1 Hypergraph based heuristic

As described in Section 2.2, the dynamic program seeks a max-cost flow to the sink in the acyclic hypergraph, where vertices correspond to plate status and hyperarcs correspond to combinations of them. Thus a cutting pattern can be represented by a set of hyperarcs and a set of vertices. As a first step of the heuristic, the unbounded dynamic program is run. It allows to compute the best value $U(v)$ associated with each vertex $v \in \mathcal{V}^0$ (*i.e.* the dynamic programming value associated to the corresponding state). The value $V(a)$ of a hyperarc $a \in \mathcal{A}^0$ is simply the sum of the cost of its tails:

$$V(a) = \sum_{v \in \mathcal{T}(a)} n_a(v) U(v)$$

Let define as $\mathbf{y}(\mathcal{A}') \in \mathbb{Z}_+^{|\mathcal{I}|}$ the partial solution corresponding to the source vertices in the tail sets of hyperarcs in multiset $\mathcal{A}'$:

$$\mathbf{y}_i(\mathcal{A}') = \mid \mathcal{A}' \cap \mathcal{A}(i) \mid, \forall i \in \mathcal{I}$$

The constructive stage of the first heuristic is run for a given hyperarc $a' \in \mathcal{A}^0$ and a given partial solution $\mathbf{y}' \in \mathbb{Z}_+^{|\mathcal{I}|}, \mathbf{y}' \leqslant \mathbf{D}$. The heuristic starts by adding $\mathbf{y}(\{a'\})$ to $\mathbf{y}'$. At any time, the current set $\mathcal{K}$ of *open vertices*, initialized with all non-source vertices in the tail set of hyperarc a'. In every iteration of the algorithm, a vertex v' is selected in $\mathcal{K}$ and removed from it. Then a hyperarc $a \in \Gamma^-(v)$ incoming to v' with the best value such that $\mathbf{y}' + \mathbf{y}(\{a\}) \leqslant \mathbf{D}$ is chosen, all non-source vertices in the tail set of a are added to $\mathcal{K}$, and next iteration starts. The algorithm stops when set $\mathcal{K}$ becomes empty. It returns the obtained solution $\mathbf{y}'$ and the corresponding multiset of hyperarcs $\mathcal{A}'$. The pseudocode of this heuristic is presented as Function HG-Constr-Heur.

The constructive heuristic function HG-Constr-Heur can then be embedded in a local search method. First the constructive heuristic find the best feasible solution among all hyperarc $a' \in \Gamma^-(t)$ incoming to the sink node $t \in \mathcal{V}^0$, and best solution $(y^*, \mathcal{A}^*)$ is obtained. Then the following local search algorithm is applied to this solution. At every iteration, each hyperarc $\hat{a} \in \mathcal{A}'$ is replaced

Function HG-Constr-Heur(a', y')

$\mathcal{K} \leftarrow \varnothing$; $\mathcal{A}' \leftarrow \varnothing$;
if $\mathbf{y}' + \mathbf{y}(\{a'\}) \leqslant \mathbf{D}$ **then**
  $\mathbf{y}' \leftarrow \mathbf{y}' + \mathbf{y}(\{a'\})$; $\mathcal{A}' \leftarrow \{a'\}$; $\mathcal{K} \leftarrow \mathcal{K} \cup \{\mathcal{T}(a')\backslash\mathcal{I}\}$;
while $\mathcal{K} \neq \varnothing$ **do**
  Pick $v \in \mathcal{K}$; $\mathcal{K} \leftarrow \mathcal{K}\backslash\{v\}$;
  $\mathcal{A}'' \leftarrow \{a \in \Gamma^-(v) : \mathbf{y}' + \mathbf{y}(\{a\}) \leqslant \mathbf{D}\}$;
  if $\mathcal{A}'' \neq \varnothing$ **then**
    $a'' \leftarrow \operatorname{argmax}_{a \in \mathcal{A}''} \{V(a)\}$;
    $\mathbf{y}' \leftarrow \mathbf{y}' + \mathbf{y}(\{a''\})$; $\mathcal{A}' \leftarrow \mathcal{A}' \cup \{a''\}$; $\mathcal{K} \leftarrow \mathcal{K} \cup \{\mathcal{T}(a'')\backslash\mathcal{I}\}$;
return $(\mathbf{y}', \mathcal{A}')$

by another hyperarc $\tilde{a} \in \Gamma^-(\mathcal{H}(\hat{a})), \tilde{a} \neq \hat{a}$ incoming to the same vertex as $\hat{a}$, and solution $\mathbf{y}'$ is modified accordingly. If an improved solution is found, next iteration starts. The process stops when no improvement occurred or the iteration number limit is reached (minimum between $|\mathcal{I}|$ and 50). The formal presentation of this heuristic is given as Algorithm 5.

2.5.2 Evolutionary heuristic

The second heuristic is an evolutionary algorithm inspired from Hadjiconstantinou and Iori [35]. Purpose is to start from an initial population (representing solutions to a given optimization problem) and then combine them together in order to obtain better individuals (better solutions). This type of metaheuristic is inspired from the evolutionary theory.

Let $\tilde{\mathcal{I}}$ be an unary representation of item set $i \in \mathcal{I}$, *i.e.* d_i copies of each item i are created. In a classical evolutionary algorithm, one needs to define an encoding to represent a valid solution. Such individual or genome g is represented by a sequence of integers $g_1, \ldots, g_{n(g)}$, each of them refers to the g_j-th item to cut in $\tilde{\mathcal{I}}$. Size $n(g)$ of a genome g is usually smaller than $|\tilde{\mathcal{I}}|$, since, generally, not all item copies fit into the bin (W, H). To obtain the solution and its value from a genome, the genome is decoded with the First-Fit heuristic.

Purpose of the First-Fit heuristics is to cut items by their order given by genome g. Let $\mathcal{X}$ be the stack of available plates initialized with initial plate (W, H). First-Fit heuristic takes the first available plate in $\mathcal{X}$, removes it from $\mathcal{X}$, and then tries to cut the first available item of index j in the order given by the genome g. Since guillotine cuts are considered here, this implies that this cut always divides the current plate r into two smaller subplates r' and r'' which are then added to stack $\mathcal{X}$. Note that if no remaining item in the genome fits in a given plate r, it is allowed to look among other available items to fill this plate, thus increasing the solution quality. This is related to the fact

Algorithm 5: Hypergraph based heuristic

h $(\mathbf{y}^*, \mathcal{A}^*) = (\mathbf{0}, \varnothing)$;
for $a' \in \Gamma^-(t)$ **do**
  $(\mathbf{y}, \mathcal{A}) \leftarrow$ HG-Constr-Heur$(a', \mathbf{0})$;
  if $\sum_{i\in\mathcal{I}} e_i \mathbf{y}_i > \sum_{i\in\mathcal{I}} e_i \mathbf{y}_i^*$ **then** $(\mathbf{y}^*, \mathcal{A}^*) = (\mathbf{y}, \mathcal{A})$
$k \leftarrow 0$;
repeat
  $improve \leftarrow false$; $k \leftarrow k + 1$;
  $(\mathbf{y}', \mathcal{A}') = (\mathbf{y}^*, \mathcal{A}^*)$;
  for $\hat{a} \in \mathcal{A}'$ **do**
    Let $\hat{\mathcal{A}}$ be the part of solution (flow) $\mathcal{A}'$ coming to $\mathcal{H}(\hat{a})$;
    for $\tilde{a} \in \Gamma^-(\mathcal{H}(\hat{a})), \tilde{a} \neq \hat{a}$ **do**
      $(\tilde{\mathbf{y}}, \tilde{\mathcal{A}}) \leftarrow$ HG-Constr-Heur$\big(\tilde{a}, \mathbf{y}' - \mathbf{y}(\hat{\mathcal{A}})\big)$;
      if $\sum_{i\in\mathcal{I}} e_i(\mathbf{y}'_i - \mathbf{y}_i(\hat{\mathcal{A}}) + \tilde{\mathbf{y}}_i) > \sum_{i\in\mathcal{I}} e_i \mathbf{y}_i^*$ **then**
        $\mathbf{y}^* \leftarrow \mathbf{y}' - \mathbf{y}(\hat{\mathcal{A}}) + \tilde{\mathbf{y}}$; $\mathcal{A}^* \leftarrow \mathcal{A}' \backslash \hat{\mathcal{A}} \cup \tilde{\mathcal{A}}$;
        $improve \leftarrow true$;
until $\neg improve$ **or** $k = \min\{|\mathcal{I}|, 50\}$;
Store solution $(\mathbf{y}^*, \mathcal{A}^*)$;

that the genome size is smaller than the number of total items. If no items fit into a subplate, it is discarded. The process stops when $\mathcal{X} = \varnothing$.

Once one has the decoding heuristic for a given genome g, new population has to be created. Idea is to start from an initial population and create successively new ones containing improving solutions The initial population is initialized by applying the First-Fit heuristic with $|\mathcal{I}|$ different random permutations of $\tilde{\mathcal{I}}$. To ensure good quality solutions, the initial population is then reduced using an elitist strategy. This one aims to keep the pool F of p_{size} best individuals from the initial population. Starting from this pool, a new population is created in two phases: new individual generation and crossover operation. The first phase is the same as the initial population generation. This enriches the pool by p_{size} new individuals. In the second phase, the offspring set O of individuals is produced using a two-point crossover where only 25% of offspring individuals are randomly kept.

Two-point crossover uses as input two parent individuals represented by their genomes g_1 and g_2 and create a new one g_3 by selecting parts of parent genomes. Two positions in the genome of the first parent p_1 and p_2 are randomly picked. The offspring is then created by merging genome of parent g_1 between positions 0 and p_1, genome of parent g_2 between positions p_1 and p_2 and then genome of parent g_1 between positions p_2 and $n(g_1)$.

After the crossover phase, all remaining individuals solutions are moved in

pool F. Then only the p_{size} best individuals in F are kept for the next population generation. To have enough solutions from one population generation to another, the population size p_{size} is set to 20. The total number of population generations is set to $|\mathcal{I}|/2$.

2.6 Computational experiments

To measure the efficiency of exact methods described in this chapter, computational experiments are performed both on literature instances and on real-world instances.

The first set of test problems are two-dimensional packing instances from the literature available on Beasley [5] (`http://people.brunel.ac.uk/~mastjjb/jeb/orlib/cgcutinfo.html`) and Hifi and Roucairol [42] (`ftp://cermsem.univ-paris1.fr/pub/CERMSEM/hifi/2Dcutting/2Dcutting.html`). They already have been described in Section 1.3.1. The instance set is partitioned into two datasets, named CU and CW. In CU dataset, the profit of each item is unweighted (*i.e.* equal to its area). In CW dataset, the profit of each item is weighted (*i.e.* not equal to its area). The CU dataset includes 46 instances: 2s, 3s, A1s, A2s, A3-5, STS2s, STS4s, OF1-2, W, CHL1s-4s, CHL5-7, ATP30-39, CU1-11, Hchl3s-8s. The CW dataset includes 41 instances: HH, cgcut1-3, A1-2, STS2, STS4, CHL1-4, CW1-11, ATP40-49, Hchl1-2, Hchl9, okp1-5.

The second set of instances are extracted from glass industry cutting problems. The bin size is set to $(3000, 1500)$ and $(6000, 3000)$. The number of different items $|\mathcal{I}|$ in an instance can be 50, 100 and 150. The average demand of an item is two. When the bin size is $(6000, 3000)$, the average maximum item size is around $(2000, 1100)$, its average minimum size is around $(500, 200)$. For bin size equal to $(3000, 1500)$, item maximum and minimum dimensions are divided by half. Two sets of instances are built, called A and P. In the A set, the profit of each item is unweighted; while in the P datasets the profit of each item is weighted. For P dataset, the profit of an item i is set to $\lceil \beta_i \times w_i \times h_i \rceil$ where β_i is the product of two random real numbers in interval $]0, 2]$. An instance named $A6000I50$ corresponds an instance with $B = (6000, 3000)$, $|\mathcal{I}| = 50$ and item profits are unweighted. Respectively, an instance named $P3000I150$ corresponds an instance with $B = (3000, 1500)$, $|\mathcal{I}| = 150$ and item profits are weighted. Each instance type is created 50 times. This gives a total of 300 instances for the P and also 300 instances for the A dataset.

The goal of the lead experiments is fourfold: (i) to evaluate the impact of the hypergraph preprocessing rules; (ii) to see how Lagrangian filtering performs; (iii) to compare the different exact methods; and (iv) to compare methods described here with best known approaches from the literature. To measure the impact of hypergraph building rules, this section is split in three parts. The first one compares all hypergraph simplifications discussed in Sec-

tions 2.1.3.1-2.1.3.5. Once the best configuration is found, experiments are done to evaluate the efficiency of both Lagrangian cost filtering and exact methods. The second part starts from the results of the first one and compare them with hypergraphs built using pattern enumeration from Section 2.1.3.6. This choice is motivated by the fact that the pattern enumeration is not a trivial hypergraph simplification compared to the other. It will be shown that it has an impact on the behaviour of Lagrangian cost filtering and exact methods also. The last part compares exact methods detailed here with best known literature approaches.

All experiments are done on a 2.5 Ghz Haswell Intel Xeon E5-2680 with 128Go of RAM. To solve linear programs, CPLEX 12.6. solver is used. The running time limit for each instance is one hour.

2.6.1 Computational experiments for basic hypergraph simplifications

The aim of this section is to measure the impact on the hypergraph size when this one is built with different simplifications rules explained in Sections 2.1.3.1-2.1.3.5. Once a hypergraph building configuration is found, a comparison of the different way to perform Lagrangian cost filtering is outlined. This is then followed by a comparison of exact methods.

2.6.1.1 Hypergraph simplifications

In this section, the impact of the hypergraph size reduction techniques of Section 2.1.3 is analysed. The purpose is to measure variations of hypergraph size depending on simplification rules.

Let ($init$) define the initial hypergraph size associated with dynamic programming (2.1)-(2.5) and (2.8). Configuration (SB) is the hypergraph built with configuration ($init$) and using symmetry breaking techniques from Section 2.1.3.2. Configuration (PR) represents the hypergraph built with configuration (SB) and using plate reductions from Section 2.1.3.3 and Section 2.1.3.4. Finally configuration (VS) indicates the hypergraph build with configuration (PR) and vertex smoothing technique from Section 2.1.3.5.

In Table 2.1 and for each dataset, the geometric means on the number of vertices $\mathcal{V}^0$ and hyperarcs $\mathcal{A}^0$ are reported for the column related to build the hypergraph with configuration $init$. For columns related to configurations SB,PR and VS, the percentages of reduction from initial hypergraph size are reported instead. The geometric mean on hypergraph building time t (in seconds) for all configurations is given in the right-hand part of the table. All reported times are rounded-up.

	$\mathcal{V}^0$				$\mathcal{A}^0$				t			
Dataset	*init*	SB	PR	VS	*init*	SB	PR	VS	*init*	SB	PR	VS
CU	3,911	0%	46%	53%	36,714	37%	53%	53%	0.1	0.1	0.1	0.1
CW	4,766	0%	48%	55%	45,528	36%	51%	52%	0.1	0.1	0.1	0.1
A1500I50	48,361	0%	58%	60%	1,100,534	50%	67%	67%	0.4	0.3	0.4	0.4
A1500I100	132,371	0%	50%	54%	5,018,939	50%	62%	62%	1.5	0.9	1.2	1.6
A1500I150	222,187	0%	46%	50%	11,193,590	50%	58%	58%	3.5	1.9	2.7	3.8
A3000I50	75,831	0%	64%	66%	1,844,530	52%	72%	72%	0.8	0.6	0.6	0.7
A3000I100	225,238	0%	56%	58%	9,280,704	53%	68%	68%	3.1	1.7	2.2	2.9
A3000I150	385,223	0%	50%	52%	21,885,919	53%	64%	64%	7.3	3.7	4.9	7.3
P1500I50	49,160	0%	57%	60%	1,155,488	49%	66%	66%	0.4	0.3	0.4	0.4
P1500I100	135,727	0%	49%	52%	5,250,292	50%	61%	61%	1.6	1.0	1.3	1.7
P1500I150	231,862	0%	45%	48%	11,861,637	50%	57%	57%	3.7	2.0	2.8	4.0
P3000I50	68,116	0%	67%	68%	1,565,716	52%	72%	72%	0.7	0.5	0.6	0.7
P3000I100	217,716	0%	56%	58%	8,679,053	53%	68%	69%	2.9	1.6	2.0	2.7
P3000I150	386,981	0%	50%	53%	21,623,473	53%	64%	64%	7.0	3.6	4.8	7.2

Table 2.1: Hypergraph creation results for all datasets

Observe that on average the number of hyperarcs decreases by half when the symmetry breaking (SB) rule is used. Since less hyperarcs are created, the hypergraph building time decreases. Observe that for large instances with $B = (6000, 3000)$ and $|\mathcal{I}| = 150$, even if symmetry breaking is used, there are still millions of hyperarcs in the hypergraph. Hence the symmetry breaking rule is a key asset to create smaller hypergraph in a shorter amount of time.

When the plate size reduction (PR) is used, both vertices and hyperarcs are not created. Remark that the number of vertices is reduced on average by half. Using configuration (PR) implies to do some preprocessing before building the hypergraph. This explains with the building time increases compare to configuration (SB). However it remains smaller that configuration (*init*).

Finally the vertex smoothing rule (VS) does not contribute to reduce the hypergraph size. Indeed the number of vertices decreases but the number of hyperarcs is nearly the same as when using configuration (PR).

Nevertheless hypergraph building is a one time operation and it is worthwhile to work at obtaining the most compact one as hypergraph traversals are performed many times during Lagrangian cost filtering or labelling algorithms. In the light of those results, all further experiments are realized with hypergraphs built using configuration (VS).

2.6.1.2 Lagrangian cost filtering procedure

This section establishes a comparison of Lagrangian filtering methods presented in Section 2.3 on the initial hypergraph G^0. Let notation (cg) (resp.

(rcg)) refers to the way to compute Lagrangian multipliers using column generation (resp. column-and-row generation). The subgradient is not discussed here since produced implementation was not enough effective. Results are given in Table 2.2. For each dataset, the average gap (gap) between the dual bound after column generation and a primal bound is first reported. The latter is computed with the hypergraph heuristic from Section 2.5.1 for CU and A datasets and with the evolutionary algorithm from Section 2.5.2 for CW and P datasets. In the left part of the table, for filtering procedures (cg) and (rcg), the average percentages of filtered hyperarcs are given when filtering is performed only in preprocessing with null multipliers ($\pi^0 = 0$), when filtering occurs in preprocessing and with optimal multipliers (π^*) and thirdly when filtering is applied for each multipliers ($\forall\pi$). During column generation and row-and-column generation after solving the unbounded dynamic program for a given multipliers π, a heuristic starts to check if the dynamic programming solution is feasible and improving compare to the current best known primal solution. In the right-hand part of the table, the average total time required to filter the hypergraph is reported for each way to filter it. Notation (t_{pp}) denotes the average time required to build the hypergraph and to obtain a primal bound. All reported times are rounded-up. In column generation and row-and-column generation, linear programs are solved by CPLEX 12.6.

		% filtered hyperarcs					t					
			cg		rcg				cg		rcg	
Dataset	gap	π^0	π^*	$\forall\pi$	π^*	$\forall\pi$	t_{pp}	π^0	π^*	$\forall\pi$	π^*	$\forall\pi$
CU	1.8%	50	54	55	54	54	0.1	0.1	0.1	0.1	0.1	0.1
CW	3.4%	37	52	54	52	54	0.1	0.1	0.1	0.1	0.1	0.1
A1500I50	0.8%	66	68	68	68	68	0.5	0.1	0.2	0.2	0.2	0.2
A1500I100	0.4%	83	84	85	85	85	1.8	0.1	0.4	0.7	0.5	0.7
A1500I150	0.2%	88	88	88	88	88	3.9	0.2	0.9	1.5	1.1	1.6
A3000I50	0.8%	63	64	65	64	65	0.8	0.1	0.2	0.3	0.3	0.3
A3000I100	0.4%	82	82	82	82	82	3.1	0.2	0.7	1.1	0.8	1.3
A3000I150	0.2%	90	90	91	90	91	7.8	0.4	1.4	2.5	1.8	2.7
P1500I50	4.6%	2	43	46	43	45	0.6	0.1	0.4	0.5	0.4	0.5
P1500I100	4.8%	4	55	57	55	57	2.6	0.2	1.4	2.1	1.3	1.9
P1500I150	5.5%	8	63	65	63	64	6.6	0.3	3.5	5.6	3	4.5
P3000I50	4.8%	3	46	49	46	48	0.9	0.1	0.4	0.5	0.4	0.5
P3000I100	5.1%	8	65	67	65	67	3.7	0.2	1.8	2.6	1.6	2.3
P3000I150	5.5%	7	62	64	62	63	9.9	0.5	5.1	7.9	4.3	6.4

Table 2.2: Percentage of filtered hyperarcs for each filtering procedurewhen hypergraph is built with configuration (VS)

From results in Table 2.2, it is enough to perform only one filtering pass (π^0)

on CU and A datasets. Indeed with such configuration, half of the hypergraph is filtered in a small computation time. For CW and P datasets, it becomes relevant to perform filtering with optimal multipliers value or with different multiplier values (π^*, $\forall\pi$). But, performing filtering with different multiplier values ($\forall\pi$) does not improve considerably the number of filtered hyperarcs compared to performing filtering with only optimal multipliers (π^*). It also increases the computation time.

For CU and A datasets, using (cg) or (rcg) does not increase the number of filtered hyperarcs compare to using null multipliers only. For CW and P datasets, (cg) and (rcg) methods have equivalent computation time but because of slow convergence of the first one, (rcg) method is preferred.

In summary, using all multipliers ($\forall\pi$) to filter the initial hypergraph G^0 is rarely useful; filtering with null multipliers (π^0) is enough to get a reduced hypergraph for CU and A datasets; while when working on CW and P datasets, it is best to use (π^*) and compute it with row-and-column generation (rcg). In the sequel, hypergraph G^0 is preprocessed according to these conclusions.

2.6.1.3 Exact methods

In this section, exact methods described in this chapter are compared to each other. The aim is to look at their efficiencies. In Table 2.3, (mip) refers to solving the problem directly by feeding the ILP formulation given in Section 2.2 to the MIP solver of CPLEX 12.6. Notation (rls) (resp. (dls)) refers to solving the problem with the iterative labelling algorithm without (resp. with) strong dominance check. The table is divided in two parts. On the left one, the reported results are obtained without performing Lagrangian cost filtering. The right part reports the same results but filtering is allowed in preprocessing and in labelling algorithms. For each part, the geometric mean on the preprocessing time (t_{pp}) and the geometric mean on the time required to solve a given dataset are reported ($tt_{mip}, tt_{dls}, tt_{rls}$). Reported times are in seconds and round-up. When an instance is not solved within the one hour time limit, the time limit value is used to calculation of reported values.

When using iterative labelling, a basic heuristic looks for an improving incumbent solution among generated labels after each hypergraph building. At the end of every iteration m, this heuristic retrieves the solutions corresponding to all labels $L \in \mathcal{L}^m(t^0)$, t^0 being the sink of G^0, such that p_L is greater than the current primal (lower) bound. If a solution from this set is feasible, the primal bound is updated. Note that if the solution is associated to an optimal label (*i.e.* the one of maximum profit in $\mathcal{L}^m(t^0)$) is feasible, then this solution is optimal for the initial problem. Note also that initial hypergraph G^0 was preprocessed with Lagrangian filtering with π^0 (resp. π^*) for CU and A datasets (resp. CW and P datasets). During iterations of dls and rls, only filtering with $\pi = 0$ was used.

	no filtering				filtering			
Dataset	t_{pp}	tt_{mip}	tt_{dls}	tt_{rls}	t_{pp}	tt_{mip}	tt_{dls}	tt_{rls}
CU	0.1	4.1	0.5	1.9	0.1	0.3	0.1	0.1
CW	0.1	6.0	0.4	0.7	0.2	0.8	0.2	0.2
A1500I50	0.5	219.9	45.4	1077.3	0.5	21.9	2.2	1.6
A1500I100	1.6	1884.6	64.6	1042.7	1.9	53.6	5.9	3.7
A1500I150	3.9	3175.3	138.3	1957.3	4.3	75.5	11.5	7.4
A3000I50	0.8	464.2	71.0	1221.1	0.9	43.7	4.4	3.6
A3000I100	3.2	2654.7	185.8	1852.3	3.4	125.0	11.1	7.8
A3000I150	7.8	3419.1	370.2	2440.5	8.4	221.3	22.1	13.4
P1500I50	0.5	522.5	282.9	341.2	1.1	97.5	44.2	24.9
P1500I100	1.8	2302.5	400.7	544.4	4.4	279.3	60.8	44.2
P1500I150	4.0	3418.6	988.2	1624.0	10.3	1262.7	90.9	64.7
P3000I50	0.7	637.3	280.1	342.9	1.4	141.2	43.0	25.3
P3000I100	2.9	2663.4	381.5	552.4	5.8	344.2	51.9	41.0
P3000I150	7.7	3478.6	872.1	1358.7	15.5	877.5	160.6	144.2

Table 2.3: Solving time for each exact methods

When the filtering is not used, computation time are very high for A and P datasets. In this case, the best method to use is the labelling algorithm with strong dominance check (dls). It seems better to use Lagrangian cost filtering since it greatly improves the computation time. Although extra computation time is required to obtain an incumbent solution and to perform filtering, the number of filtered hyperarcs leads to an advantage for the exact methods. For CU and CW datasets, both exact methods are competitive since computation time is short. For A and P datasets, it is better to use labelling algorithms. The best one to use is the one without strong dominance check (rls). The ILP formulation has the worst results and it is clearly not the best approach to retain.

Beyond computation time comparison, it is interesting to count the number of solved instances by each method. Such results are reported in Table 2.4. Notations nb_s, nb_f and nb_{ns} are the number of solved instances for a given method, the number of cases where the method is the fastest and the number of instances that are not solved by any of the methods.

When no filtering occurs, the method dls solves most of the problem instances for A dataset. The ILP formulation mip is able to solve most of instances only for datasets with a small number of items. Results are not very conclusive on P datasets because between one and two thirds of the instances are not solved. Without Lagrangian cost filtering, the method dls is the fastest method to use. Nevertheless, when filtering is used, the method rls is preferred. Note also that filtering increases the number of solved instances for all exact methods. Labelling algorithms are efficient to solve the problem

	no filtering							filtering						
	mip		*dls*		*rls*			*mip*		*dls*		*rls*		
Dataset	nb_s	nb_f	nb_s	nb_f	nb_s	nb_f	nb_{ns}	nb_s	nb_f	nb_s	nb_f	nb_s	nb_f	nb_{ns}
CU (46)	46	7	42	28	35	11	0	46	2	44	6	44	31	0
CW (41)	41	2	39	25	37	14	0	41	0	41	3	41	31	0
A1500I50 (50)	48	11	44	38	10	0	1	48	0	50	7	50	43	0
A1500I100 (50)	23	0	50	50	11	0	0	50	0	50	1	50	49	0
A1500I150 (50)	8	0	49	49	8	0	1	44	0	50	1	50	49	0
A3000I50 (50)	46	7	41	40	9	0	3	49	1	50	13	50	35	0
A3000I100 (50)	15	1	41	40	7	1	8	42	0	50	7	50	43	0
A3000I150 (50)	5	0	47	47	4	0	3	44	0	50	0	50	50	0
P1500I50 (50)	39	19	25	15	23	5	11	40	3	36	0	41	38	7
P1500I100 (50)	20	0	25	22	21	3	25	28	0	39	0	40	39	10
P1500I150 (50)	4	1	22	21	11	1	27	23	0	46	8	47	41	1
P3000I50 (50)	38	16	28	16	21	7	11	40	2	39	1	44	42	4
P3000I100 (50)	16	0	27	24	23	3	23	30	0	44	5	44	40	5
P3000I150 (50)	3	0	21	19	13	2	29	18	0	37	8	38	32	10

Table 2.4: Number of solved instances for all exact methods

contrary to the MILP approach.

To further analyse the performance of labelling algorithms, it is interesting to have a look on the number of labels and hyperarcs created during the solution process. Results are reported in Table 2.5. For each dataset, the number of instances (nb_s) solved by both labelling algorithms is first reported. Note that this number is smaller than in Table 2.4 since it may happen that an instance is solved in preprocessing when incumbent solution is equal to the dual bound from filtering. The comparison of Table 2.5 carries on those solved instances. The geometric mean of the number of iterations (nb_{it}), the number of labels (nb_l), and the number of hyperarcs (nb_h) are reported in this table. All values are rounded-up.

As observed from Table 2.5, the *dls* method produces more labels than *rls* method but the number of hyperarcs is smaller on average. This is somehow conter-intuitive that the strong dominance produces more labels than the weak dominance. The explanation is to be found in the combination with filtering. Having more labels and thus more hyperarcs in first iterations allows one to filter more hyperarcs. Then, every additional filtered hyperarc in a "smaller dimension" iteration, filters implicitly many hyperarcs in a "larger dimension" iterations which project on the former. Thus, filtering combined with the weak dominance may be more efficient in comparison with the strong dominance. The difference between the number of iterations for both methods is explained by the heuristic used to improve current best solution. Indeed, having more labels in $\mathcal{L}^m(t^0)$ increases the probability to find a primal feasible solution.

		dls			rls		
Dataset	nb_s	nb_{it}	$nb_l(/10^5)$	$nb_h(/10^6)$	nb_{it}	$nb_l(/10^5)$	$nb_h(/10^6)$
CU	36	6.0	2.8	0.6	6.0	0.5	0.9
CW	34	4.2	0.4	0.2	4.2	0.3	0.5
A1500I50	50	16.0	18.6	5.1	16.0	5.4	16.1
A1500I100	50	19.0	18.1	7.6	19.0	3.8	7.6
A1500I150	50	21.7	23.3	12.5	21.6	5.8	16.9
A3000I50	49	16.6	44.8	12.8	16.5	14.0	32.1
A3000I100	50	21.0	27.9	13.3	21.0	9.0	24.1
A3000I150	50	22.3	28.1	15.3	22.2	6.7	13.0
P1500I50	34	7.7	33.1	106.0	7.7	13.7	95.7
P1500I100	38	7.8	42.9	145.6	7.8	18.6	184.6
P1500I150	44	8.8	49.4	125.9	8.8	24.3	189.6
P3000I50	38	7.3	46.8	126.6	7.3	22.8	123.1
P3000I100	43	7.2	47.8	146.1	7.2	25.5	221.6
P3000I150	35	8.0	57.1	138.3	8.0	28.6	188.9

Table 2.5: Number of labels and hyperarcs for both label setting algorithms

2.6.2 Computational experiments for partial pattern enumeration

Results in Section 2.6.1.1 outline the viability of the iterative labelling algorithm to solve guillotine 2KP. Nevertheless only basic hypergraph simplification has been considered. In this section, the scope is extended to measure now the impact of partial enumeration technique from Section 2.1.3.6. As previously, a first inspection of the best hypergraph configuration is done. Once retained, it is used to measure the efficiency of Lagrangian cost filtering and exact methods.

2.6.2.1 Hypergraph simplifications

In previous experiments, the configuration (VS) was retained to build the hypergraph. In this section, the enhanced pattern enumeration is compared with the configuration (VS). Remember that the configuration of the partial enumeration is characterized by four values: Δ_w^{size}, Δ_h^{size}, Δ_w^{diff}, and Δ_h^{diff}. Three settings are considered here. The first one corresponds to the hypergraph building using configuration (VS) with no enumeration: $\Delta_1 = (\Delta_w^{size} = 0, \Delta_h^{size} = 0, \Delta_w^{\text{diff}} = 0, \Delta_h^{\text{diff}} = 0)$. Notation Δ_1 is equivalent to build the hypergraph with configuration (VS). The second one corresponds to the hypergraph building using configuration (VS) and enumeration of items with the same h_i for odd cutting stages and with the same w_i for even cutting stages: $\Delta_2 = (\Delta_w^{size} = 1000, \Delta_h^{size} = 1000, \Delta_w^{\text{diff}} = 1, \Delta_h^{\text{diff}} = 1)$. The last setting corre-

sponds to the hypergraph building using configuration (VS) and enumeration of items with the same h_i for odd cutting stages and enumeration of items with different w_i at the second cutting stage: $\Delta_3 = (\Delta_w^{size} = 1000, \Delta_h^{size} = 1000, \Delta_w^{\text{diff}} = \infty, \Delta_h^{\text{diff}} = 1)$.

In Table 2.6 and for each dataset, the geometric means on the number of vertices $\mathcal{V}^0$ and hyperarcs $\mathcal{A}^0$ are reported for the column related to build the hypergraph with configuration *init*. For columns related to configurations (Δ_1), (Δ_2) and (Δ_3), the percentages of reduction from initial hypergraph size are reported instead. The geometric mean on hypergraph building time t (in seconds) for all configurations is given in the right-hand part of the table. All reported times are rounded-up.

	$\mathcal{V}^0$				$\mathcal{A}^0$				t			
Dataset	*init*	Δ_1	Δ_2	Δ_3	*init*	Δ_1	Δ_2	Δ_3	*init*	Δ_1	Δ_2	Δ_3
CU	3,911	53%	21%	32%	36,714	53%	45%	33%	0.1	0.1	0.1	0.1
CW	4,766	55%	20%	32%	45,528	52%	44%	34%	0.1	0.1	0.1	0.1
A1500I50	48,361	60%	47%	50%	1,100,534	67%	66%	61%	0.4	0.4	0.4	0.5
A1500I100	132,371	54%	36%	38%	5,018,939	62%	60%	54%	1.5	1.6	1.2	2
A1500I150	222,187	50%	30%	32%	11,193,590	58%	56%	51%	3.5	3.8	2.6	4.4
A3000I50	75,831	66%	56%	58%	1,844,530	72%	71%	68%	0.8	0.7	0.6	0.7
A3000I100	225,238	58%	45%	47%	9,280,704	68%	67%	63%	3.1	2.9	2.2	3.1
A3000I150	385,223	52%	38%	40%	21,885,919	64%	62%	58%	7.3	7.3	4.9	7
P1500I50	49,160	60%	45%	48%	1,155,488	66%	64%	60%	0.4	0.4	0.4	0.5
P1500I100	135,727	52%	34%	37%	5,250,292	61%	59%	54%	1.6	1.7	1.3	2.1
P1500I150	231,862	48%	29%	31%	11,861,637	57%	55%	50%	3.7	4.0	2.7	4.7
P3000I50	68,116	68%	58%	60%	1,565,716	72%	71%	68%	0.7	0.7	0.6	0.7
P3000I100	217,716	58%	46%	48%	8,679,053	69%	67%	64%	2.9	2.7	2.1	2.9
P3000I150	386,981	53%	39%	40%	21,623,473	64%	63%	59%	7.0	7.2	4.8	6.7

Table 2.6: Hypergraph creation results for all datasets

From Table 2.6, it is clear that using the partial enumeration with (Δ_2) and (Δ_3) increases the hypergraph size compare to configuration (Δ_1). Building time has the same order of magnitude that using configuration (*init*). In theory the enumeration enforces item production constraints directly by modifying the hypergraph structure. It may seem odd to keep configuration (Δ_3) but the hope is to strengthen the item production constraints. Thus, more experiments are required to ensure that it is worth to enumerate pattern. This is outlined in the next section.

2.6.2.2 Lagrangian cost filtering procedure

In Section 2.6.2.1, using the configuration (Δ_3) to build the hypergraph G^0 seems strange because it leads to a bigger hypergraph compare to using the

configuration (VS). In this section, the impact of the partial pattern enumeration is outlined by applying Lagrangian cost filtering on the hypergraph built with configuration (Δ_3). Results are reported in Table 2.7. Experiments are done in the same manner as described in Section 2.6.1.2. Consequently same notations are reused here and Table 2.7 has the same structure that Table 2.2. As a reminder, before starting filtering, an initial primal bound is obtained with the hypergraph heuristic from Section 2.5.1 for CU and A datasets and with the evolutionary algorithm from Section 2.5.2 for CW and P datasets. During column generation and row-and-column generation after solving the unbounded dynamic program for a given multipliers π, a heuristic starts to check if the dynamic programming solution is feasible and improving compare to the current best known primal solution. All reported times are rounded-up. In column generation and row-and-column generation, linear programs are solved by CPLEX 12.6.

		% filtered hyperarcs					t					
			cg		rcg				cg		rcg	
Dataset	gap	π^0	π^*	$\forall\pi$	π^*	$\forall\pi$	t_{pp}	π^0	π^*	$\forall\pi$	π^*	$\forall\pi$
CU	0.6%	93	95	95	95	95	0.2	0.1	0.1	0.1	0.1	0.1
CW	2.7%	63	74	75	74	75	0.2	0.1	0.1	0.1	0.1	0.1
A1500I50	0.2%	96	97	97	97	97	0.6	0.1	0.1	0.2	0.2	0.2
A1500I100	0.1%	99	99	99	99	99	2.2	0.2	0.3	0.5	0.5	0.6
A1500I150	0.1%	99	99	99	99	99	5	0.3	0.6	0.9	0.8	1.1
A3000I50	0.2%	94	95	95	95	95	0.9	0.1	0.2	0.3	0.2	0.3
A3000I100	0.1%	99	99	99	99	99	3.4	0.2	0.5	0.8	0.7	0.9
A3000I150	0.1%	99	99	99	99	99	7.6	0.5	1	1.6	1.3	1.9
P1500I50	3.9%	17	56	58	56	57	0.8	0.1	0.9	1.3	0.6	0.9
P1500I100	4.1%	25	67	68	67	68	3.2	0.2	3	4.7	2.1	3.3
P1500I150	4.8%	28	75	77	75	76	7.4	0.5	5.3	8.1	3.8	5.8
P3000I50	4.1%	10	54	57	54	56	1	0.1	0.9	1.4	0.8	1.1
P3000I100	4.4%	30	72	74	72	73	4.1	0.3	3.4	5.6	2.7	3.9
P3000I150	4.8%	24	70	72	70	72	9.8	0.7	9.1	14.1	6.6	10.5

Table 2.7: Percentage of filtered hyperarcs for each filtering procedure when hypergraph is built with configuration (Δ_3)

From results in Table 2.7, the same conclusion when hypergraph is built with configuration (Δ_3) or (VS) is done about filtering The Lagrangian cost filtering with null multipliers (π^0) is enough to get a reduced hypergraph for CU and A datasets. It is best to use (π^*) and compute it with row-and-column generation (rcg) for CW and P datasets. Nevertheless, it is important to remark that filtering with null multipliers removes near all hyperarcs for CU and A datasets. The gap between primal and dual bounds is close to

zero. Results for CW and P datasets outline that the initial filtering with null multipliers remove more hyperarcs compare to results from Table 2.2.

A comparison of hypergraph size after filtering is given in Table 2.8. Notation $|\mathcal{A}^0|$ is the round-up geometric mean on the number of hyperarcs, notation $|\tilde{\mathcal{A}}^0|$ is the round-down geometric mean on number of hyperarcs after filtering. Reported time t (in seconds) is the round-up geometric mean required to build the hypergraph and to compute a primal bound. Each measure is done for configurations (VS) and (Δ_3).

	VS			Δ_3		
Dataset	$\lvert\mathcal{A}^0\rvert$	$\lvert\tilde{\mathcal{A}}^0\rvert$	t	$\lvert\mathcal{A}^0\rvert$	$\lvert\tilde{\mathcal{A}}^0\rvert$	t
CU	17277	8578	0.2	115017	7130	0.3
CW	21936	10409	0.2	115107	29294	0.3
A1500I50	366588	123795	0.6	509107	16398	0.7
A1500I100	1949677	317038	1.9	2635457	14716	2.4
A1500I150	4753812	553348	4.1	6116738	7244	5.3
A3000I50	521408	189364	0.9	721779	37797	1
A3000I100	3015578	534710	3.3	3938741	22982	3.6
A3000I150	8049202	753518	8.2	10192619	26529	8.1
P1500I50	396350	223001	1	556820	243200	1.4
P1500I100	2082839	921163	3.9	2739695	902371	5.3
P1500I150	5104638	1877492	9.6	6383063	1559034	11.2
P3000I50	439947	235735	1.3	619052	282697	1.8
P3000I100	2774874	955921	5.3	3702391	1012000	6.8
P3000I150	7844847	2957148	14.2	10000778	2903615	16.4

Table 2.8: Number of filtered hyperarcs for configurations (VS) and (Δ_3)

For CU and A datasets, the effect of Lagrangian cost filtering for configuration (VS) reduces the hypergraph size by half as mentioned previously. Using configuration (Δ_3), the reduction is more important and the hypergraph after filtering is smaller than the one obtained from configuration (VS). For CW and P datasets, it is not clear if using one configuration dominates the other. Indeed after Lagrangian cost filtering, the sizes of filtered hypergraphs have the same order of magnitude. Computation times on both configurations are equivalent.

2.6.2.3 Exact methods

From Section 2.6.2.1 and Section 2.6.2.2, using the hypergraph configuration (Δ_3) allows one to obtain a filtered hypergraph of equivalent size compare to using configuration (VS). In this section, exact methods for hypergraph build with configuration (Δ_3) are compare to each other. Results are then compared

with the results from Section 2.6.1.3. Tables 2.9-2.11 have the same structure that Tables 2.3-2.5. Reported values are also done in the same way.

	no filtering				filtering			
Dataset	t_{pp}	tt_{mip}	tt_{dls}	tt_{rls}	t_{pp}	tt_{mip}	tt_{dls}	tt_{rls}
CU	0.1	4.1	0.2	0.6	0.0	0.1	0.1	0.1
CW	0.1	6.2	0.3	0.5	0.1	0.5	0.2	0.2
A1500I50	0.5	181.4	5.6	82.6	0.6	1.5	0.7	0.6
A1500I100	2.1	1852.4	9.6	94.1	2.4	3.7	2.8	2.5
A1500I150	4.7	3312.6	22.6	141.2	5.3	7.0	6.2	5.6
A3000I50	0.7	343.6	7.2	107.3	0.9	3.3	1.2	1.0
A3000I100	3.2	2782.5	22.6	217.0	3.6	6.6	4.4	3.9
A3000I150	7.4	3583.3	49.8	343.0	8.3	13.3	10.0	8.9
P1500I50	0.7	355.3	166.8	191.6	1.2	51.4	32.7	18.3
P1500I100	3.1	2152.4	254.8	344.3	4.9	163.8	44.0	33.9
P1500I150	7.8	3436.3	662.8	1075.5	11.5	624.1	61.8	49.2
P3000I50	0.9	477.6	195.2	225.8	1.5	75.1	34.9	18.3
P3000I100	4.1	2591.3	215.3	297.6	6.1	185.4	37.9	28.5
P3000I150	9.8	3569.3	529.2	736.5	15.9	660.3	115.7	95.8

Table 2.9: Solving time for each exact methods

From results in Table 2.9, the same conclusions are drawn that from Table 2.3. Nevertheless, the main difference between the two tables is the computation time required for each method. Indeed without filtering, the average computation time to solve instances is smaller with configuration (Δ_3) compared to configuration (VS). When filtering occurs, the computation time decreases again.

In Table 2.10, the new hypergraph configuration increases the number of solved instances with filtering compare to results in Table 2.4. When no filtering, the partial pattern enumeration benefits to labelling algorithms. Using the configuration (Δ_3) allows one to solve more problem instances on average.

From the comparison of Tables 2.5 and 2.11, the application of the partial pattern enumeration leads to perform less iterations on average in labelling algorithms. At the same time, the average number of labels and hyperarcs becomes smaller.

As outlined in previous tables, the usage of partial pattern enumeration improves the resolution of the problem. This seems logical since the partial pattern enumeration enforces item production constraints directly in the hypergraph representation. To establish a fair comparison, the best results from Section 2.6.1.3 and the one presented here are compared hereinafter. The first comparison is shown in Table 2.12 and presents the average computation time required to solve all datasets when the hypergraph is built with configurations

	no filtering							filtering						
	mip		*dls*		*rls*			*mip*		*dls*		*rls*		
Dataset	nb_s	nb_f	nb_s	nb_f	nb_s	nb_f	nb_{ns}	nb_s	nb_f	nb_s	nb_f	nb_s	nb_f	nb_{ns}
CU (46)	46	4	44	28	39	14	0	46	2	44	2	44	27	0
CW (41)	41	2	39	22	38	17	0	41	0	41	1	41	34	0
A1500I50 (50)	50	2	49	47	32	1	0	50	0	50	2	50	47	0
A1500I100 (50)	26	0	50	46	31	4	0	50	0	50	0	50	45	0
A1500I150 (50)	7	0	50	45	30	5	0	50	0	50	2	50	45	0
A3000I50 (50)	50	1	49	45	30	4	0	50	0	50	2	50	44	0
A3000I100 (50)	15	0	50	47	30	3	0	50	0	50	0	50	48	0
A3000I150 (50)	1	0	50	43	25	7	0	50	0	50	1	50	41	0
P1500I50 (50)	41	20	28	9	25	12	9	43	6	38	0	42	36	6
P1500I100 (50)	27	2	28	21	23	7	20	38	1	40	1	40	39	8
P1500I150 (50)	5	0	25	22	14	3	25	38	0	47	10	47	38	2
P3000I50 (50)	42	18	28	11	24	13	8	42	3	39	0	47	43	2
P3000I100 (50)	20	0	30	18	23	12	20	37	0	44	4	46	41	4
P3000I150 (50)	2	0	26	20	18	6	24	24	1	38	5	39	33	10

Table 2.10: Number of solved instances for all exact methods

		dls			*rls*		
Dataset	nb_s	nb_{it}	$nb_l(/10^5)$	$nb_h(/10^6)$	nb_{it}	$nb_l(/10^5)$	$nb_h(/10^6)$
CU	29	3.6	0.2	0.1	3.5	0.1	0.1
CW	35	3.5	0.2	0.1	3.5	0.2	0.2
A1500I50	49	6.2	0.3	0.1	6.1	0.1	0.1
A1500I100	45	5.0	0.1	0.0	5.0	0.1	0.0
A1500I150	47	5.3	0.1	0.0	5.3	0.1	0.0
A3000I50	46	6.0	0.7	0.4	5.9	0.4	0.4
A3000I100	48	6.0	0.2	0.1	6.0	0.1	0.0
A3000I150	42	6.8	0.2	0.1	6.6	0.2	0.1
P1500I50	36	7.2	22.4	87.1	7.2	11.1	65.4
P1500I100	38	6.9	16.1	79.6	6.9	8.8	70.0
P1500I150	46	8.0	30.1	102.6	8.0	18.1	142.9
P3000I50	37	6.9	37.4	118.3	6.9	17.2	84.9
P3000I100	43	6.6	27.9	114.3	6.6	18.9	166.5
P3000I150	37	7.5	31.2	94.6	7.5	18.0	118.0

Table 2.11: Number of labels and hyperarcs for both label setting algorithms

(VS) and (Δ_3). Reported times are in seconds and rounded up. When an instance is not solved within the one hour time limit, the time limit value is used to calculation of reported values.

The configuration (Δ_3) greatly contributes to reducing the computation

	VS			Δ_3		
Dataset	tt_{mip}	tt_{dls}	tt_{rls}	tt_{mip}	tt_{dls}	tt_{rls}
CU	0.3	0.1	0.1	0.1	0.1	0.1
CW	0.7	0.2	0.1	0.5	0.2	0.2
A1500I50	21.8	2.1	1.6	1.5	0.7	0.6
A1500I100	53.6	5.8	3.7	3.7	2.8	2.5
A1500I150	75.4	11.4	7.3	7.0	6.2	5.6
A3000I50	43.6	4.3	3.6	3.3	1.2	1.0
A3000I100	124.9	11.0	7.8	6.6	4.4	3.9
A3000I150	221.2	22.0	13.4	13.3	10.0	8.9
P1500I50	97.4	44.1	24.8	51.4	32.7	18.3
P1500I100	279.3	60.7	44.1	163.8	44.0	33.9
P1500I150	1262.7	90.9	64.7	624.1	61.8	49.2
P3000I50	141.2	42.9	25.3	75.1	34.9	18.3
P3000I100	344.1	51.8	40.9	185.4	37.9	28.5
P3000I150	877.4	160.5	144.2	660.3	115.7	95.8

Table 2.12: Solving time for each exact methods for configurations VS and Δ_3

time to solve the problem with (mip) especially for dataset A. For both labelling algorithms, the partial pattern enumeration also leads to smaller computation time. In conclusion even if more time is spent to build the hypergraph with (Δ_3), this globally decreases the computation time required to solve the problem. The number of instances solved in each dataset for both methods are shown in Table 2.13. Again configuration (Δ_3) contributes to increase the number of solved instances for all exact methods. The pattern enumeration benefits the most to the labelling algorithm without strong dominance (rls).

	VS							Δ_3						
	mip		dls		rls			mip		dls		rls		
Dataset	nb_s	nb_f	nb_s	nb_f	nb_s	nb_f	nb_{ns}	nb_s	nb_f	nb_s	nb_f	nb_s	nb_f	nb_{ns}
CU (46)	46	2	44	6	44	31	0	46	2	44	2	44	27	0
CW (41)	41	0	41	3	41	31	0	41	0	41	1	41	34	0
A1500I50 (50)	48	0	50	7	50	43	0	50	0	50	2	50	47	0
A1500I100 (50)	50	0	50	1	50	49	0	50	0	50	0	50	45	0
A1500I150 (50)	44	0	50	1	50	49	0	50	0	50	2	50	45	0
A3000I50 (50)	49	1	50	13	50	35	0	50	0	50	2	50	44	0
A3000I100 (50)	42	0	50	7	50	43	0	50	0	50	0	50	48	0
A3000I150 (50)	44	0	50	0	50	50	0	50	0	50	1	50	41	0
P1500I50 (50)	40	3	36	0	41	38	7	43	6	38	0	42	36	6
P1500I100 (50)	28	0	39	0	40	39	10	38	1	40	1	40	39	8
P1500I150 (50)	23	0	46	8	47	41	1	38	0	47	10	47	38	2
P3000I50 (50)	40	2	39	1	44	42	4	42	3	39	0	47	43	2
P3000I100 (50)	30	0	44	5	44	40	5	37	0	44	4	46	41	4
P3000I150 (50)	18	0	37	8	38	32	10	24	1	38	5	39	33	10

Table 2.13: Number of solved instances for all exact methods for configurations VS and Δ_3

2.7 Conclusion

In this chapter, a mixed integer programming and a dynamic programming based exact solution method for the two-dimensional knapsack problem are presented. The labelling algorithm for the unbounded case is shown to admit a network flow representation in a hypergraph. To handle the huge size of the hypergraph, preprocessing techniques such a partial pattern enumeration and a filtering procedure are developed. The latter fixes hyperarcs by reduced cost after a Lagrangian relaxation of the production bounds. From there, three algorithms are derived: (*i*) solving a max-cost flow formulation in the reduced size hypergraph with side constraints to enforce production bounds; (*ii*) adapting the dynamic programming recursion to the bounded case, by extending the state space dynamically and applying filtering; (*iii*) a variant of the latter where a strong dominance rule is applied at the expense of using a weaker filtering procedure. Computational experiments demonstrate the positive impact of preprocessing, filtering and dominance procedures. Obtained results are even better when partial patterns are enumerated within the hypergraph. The dynamic programs are shown to provide exact solution to relatively large size industrial instances in most cases. This chapter used the C-2KP-RE-4-r to support explanations of designed methods. Nevertheless, methods described here are generic and can be used to solve variants of guillotine 2KP.

Chapter 3

Bin-packing problems

A way to remove a part of the initial industrial problem difficulty is to not consider defects at all. This relaxed the initial problem to the 2BP$_l$. An advantage of this relaxation is that there is no need to differentiate patterns from one bin to another since all bins are identical. Defects can then be handled by using a post-processing methods of an obtained 2BP$_l$ solution. The first half of this chapter details how to solve the 2BP$_l$. In the second half, solving methods for the 2BP$_l$ are modified to be able to handle consecutive 2BP$_l$ instances. This comes a simple observation. In the cutting industry, item sets have often to be cut consecutively in a given order. This order is derived from due date of customer orders. For a given batch of items, a cutting problem is solved with leftover consideration to save raw material from the last cut bin. The saved bin part can then be used to cut the next item batch. The main motivation is to write and solve ILP formulations to describe the behavior of a factory in which different item batches have to be cut one after the other.

3.1 Solving the 2BP$_l$

In this section, the Dantzig-Wolfe reformulation of the 2BP$_l$ is first outlined. Then a diving heuristic to obtain good solutions to this problem is proposed. Finally, computational results are presented and show that the obtained heuristic solutions are indeed close to optimality.

3.1.1 Formulations of the 2BP$_l$

The standard way to solve the 2BP$_l$ is to apply Dantzig-Wolfe decomposition. The resulting ILP formulation can then be tackled with column generation and branch-and-price to obtain an optimal integer solution. Such formulation is explained in the first part of this section. According to the works of Valério de Carvalho [79], the 2BP$_l$ can also be formulated with a pseudo-polynomial size formulation. This is the second subject discussed in this section.

3.1.1.1 Standard formulation

The ILP formulation of the 2BP$_l$ is every similar to the one for the 2BP. Recall that in the 2BP$_l$, there are two subproblems to deal with. One related to use entirely a bin and one related to minimize the residual used length. Let $\mathcal{P}^1$ (resp. $\mathcal{P}^2$) be the set of valid cutting patterns for standard bins (resp. the last used bin) in a 2BP$_l$ instance. Let integer variable λ_p be the number of times a pattern p is used. Since all bins have the same size, the set of patterns $\mathcal{P}^1$ and $\mathcal{P}^2$ are valid for all bins. Notation a_{ip} is the number of items $i \in \mathcal{I}$ cut in pattern $p \in \mathcal{P}^1 \cup \mathcal{P}^2$, w_p is the total width of first stage cuts in a pattern $p \in \mathcal{P}^2$. An ILP formulation for the 2BP$_l$ is the following:

$$\min \sum_{p \in \mathcal{P}^1} W\lambda_p + \sum_{p \in \mathcal{P}^2} w_p \lambda_p \tag{3.1}$$

$$\text{s.t.} \sum_{p \in \mathcal{P}^1} a_{ip}\lambda_p + \sum_{p \in \mathcal{P}^2} a_{ip}\lambda_p = d_i, \quad \forall i \in \mathcal{I} \tag{3.2}$$

$$\sum_{p \in \mathcal{P}^2} \lambda_p = 1 \tag{3.3}$$

$$\lambda_p \in \mathbb{N}, \quad \forall p \in \mathcal{P}^1 \tag{3.4}$$

$$\lambda_p \in \{0, 1\}, \quad \forall p \in \mathcal{P}^2 \tag{3.5}$$

Objective function (3.1) ensures to minimize the total used width among bins. First part of the function is related to complete used width for standard bins. Second objective part is related to the total used width for last bin. Constraints (3.2) ensure to cut ordered items. Constraint (3.3) forces to use exactly one cutting pattern for the last bin. The number of pricing problems to solve at each column generation iteration is 2. Let π the set of dual variables associated with constraints (3.2) and ι the dual variable associated with constraint (3.3). The reduces costs of finding a pattern p are:

$$\zeta(\lambda_p) = -\sum_{i \in \mathcal{I}} a_{ip}\pi_i + \begin{cases} W, & p \in \mathcal{P}^1 \\ w_p - \iota, & p \in \mathcal{P}^2 \end{cases} \tag{3.6}$$

If one is interested in solving the 2BP instead of the 2BP$_l$, one has to remove all references to $\mathcal{P}^2$ in formulation (3.1)-(3.5). Consequently, only one subproblem remains.

The ILP formulation (3.1)-(3.5) is tackled with column generation. Pricing a new column corresponds to solve to optimality a C-2KP-RE-4-r instance. From results presented in Section 2.2 and Section 2.4, such pricing problem can be solved to optimality. Consequently, one can solve the linear relaxation of (3.1)-(3.5) with column generation and then starts a branch-and-price to obtain integer solution. Using the generic branching scheme described in

Vanderbeck [82], the subproblem has to be updated with side constraints to respect branching decisions. Nevertheless in practice and according to results from Section 2.6, the computation time required to solve only one subproblem, *i.e.* one C-2KP-RE-4-r instance, may be long. Therefore exact methods for the C-2KP-RE-4-r are not a good choice in a column generation context since it will require a huge computation time.

3.1.1.2 Pseudo-polynomial size formulation

From the works of Valério de Carvalho [79], it is possible to write a pseudo-polynomial size formulation for the 2BP$_l$. For the 1BP, author writes the 1KP as a flow problem in a directed acyclic graph. The pseudo-polynomial MILP flow formulation of this pricing problem is then rewritten to obtain a pseudo-polynomial size formulation for the 1BP. More details can be found in Section 1.2.3.2. Using the same methodology and since the C-2KP-RE-4-r can be solved to optimality with a pseudo-polynomial size formulation as outlined in Section 2.2, one can derive a pseudo-polynomial size formulation for the 2BP$_l$.

Since the 2BP$_l$ has two subproblems, let define two hypergraphs indexed by k. $k = 2$ stands for the standard bin subproblem and $k = 3$ for the last bin subproblem. Let $G_k = (\mathcal{V}_k, \mathcal{A}_k)$ be the hypergraph associated to subproblem $k \in \{2, 3\}$. Let z_k be the total number of patterns of type k that are used. Integer variables x_a represent the flow value going through hyperarc $a \in \mathcal{A}_k, k \in \{2, 3\}$. The number of times (multiplicity) of vertex $v \in \mathcal{V}_k, k \in \{2, 3\}$ is cut when choosing hyperarc $a \in \mathcal{A}_k, k \in \{2, 3\}$ is represented by notation $n_a(v)$. Notation $\mathcal{A}_k(i), k \in \{2, 3\}$, defines the multiset of hyperarcs whose tail sets include a boundary vertex representing item $i \in \mathcal{I}$. The extended pseudo-polynomial size ILP formulation for the 2BP$_l$ is given by:

$$\min W z_2 + z_3 \sum_{a \in \bar{\mathcal{A}}_3} w_a x_a \tag{3.7}$$

$$\text{s.t.} \sum_{a \in \Gamma^-(v)} x_a - \sum_{a' \in \Gamma^+(v)} n_{a'}(v) x_{a'} = 0, \quad \forall v \in \mathcal{V}_k \backslash \{t_k \cup \mathcal{I} \cup \varnothing\}, \forall k \in \{2, 3\} \tag{3.8}$$

$$\sum_{a \in \Gamma^-(t_k)} x_a = z_k, \quad \forall k \in \{2, 3\} \tag{3.9}$$

$$\sum_{k \in \{2,3\}} \sum_{a \in \mathcal{A}_k(i)} n_a(i) x_a = d_i, \quad \forall i \in \mathcal{I} \tag{3.10}$$

$$z_3 = 1, \quad z_2 \in \mathbb{N} \tag{3.11}$$

$$x_a \in \mathbb{N}, \quad \forall a \in \mathcal{A}_k, \forall k \in \{2, 3\} \tag{3.12}$$

The ILP formulation (3.7)-(3.12) is derived from the one used for the 2KP

defined in Section 2.2. Objective function (3.7) minimizes the number of patterns for standard bins and the length of the pattern selected for the last bin. The contribution to the objective function is given by the sum of the widths w_a of hyperarcs $a \in \bar{\mathcal{A}}_3$, with $\bar{\mathcal{A}}_3$ the set of hyperarcs associated to cuts performed only at the first cutting stage. Note that only one pattern can be created for the last bin subproblem according to constraint (3.11) and problem definition. Constraints (3.8) ensure flow conservation in both hypergraphs. Constraints (3.9) bound the flow quantity in both hypergraphs. Constraint set (3.10) ensure that the right amount of items $i \in \mathcal{I}$ are cut among all patterns. Note that in order to solve the 2BP only, the parts of the formulation related to $k = 3$ have to be omitted. The optimal solution of a 2BP$_l$ instance can be found by solving the formulation (3.7)-(3.12) using a MILP solver. Note that, since this formulation is based on the hypergraph representation, a huge hypergraph will induce a huge formulation which may be not solved within a reasonable amount of time.

3.1.2 Diving heuristic with non-proper columns

Using Dantzig-Wolfe decomposition or a pseudo-polynomial size formulation for the 2BP$_l$ does not seem to be a pertinent choice to solve large instances. However one can exploit the block angular structure before applying Dantzig-Wolfe decomposition and apply a diving heuristic. To achieve primal feasibility, the basic diving heuristic requires to use so-called proper columns (*i.e.* variables that could take a non-zero value in an integer solution of the residual master problem). In the 2BP$_l$, a variable $\lambda_p, p \in \mathcal{P}^1 \cup \mathcal{P}^2$ is proper if $a_{ip} \leqslant d_i, \forall i \in \mathcal{I}$. Therefore, in each pricing subproblem, there are upper bounds on the number of copies of items in the cutting pattern. In the presence of these upper bounds, the pricing problem becomes significantly harder to solve to optimality as discussed in Chapter 2. A possible solution to this issue is to solve the pricing problem only heuristically using algorithms from Section 2.5. A second approach is to use diving heuristics with non-proper columns. Cintra et al. [11] have shown that the lower bound obtained by solving the master problem with non-proper columns is close to the one obtained when using exclusively proper columns. As the quality of diving heuristics depends mainly on the strength of the master problem bound, it is expected that a "non-proper" diving heuristics will be efficient to solve the 2BP$_l$.

3.1.2.1 "Non-proper" diving heuristic

The proposed "non-proper" diving heuristic proceeds as the standard diving heuristic previously detailed in Section 1.2.4. Remember that at each diving iteration, the residual master problem (RMP) is solved by column generation. In the proposed modified diving heuristic, columns are priced thanks to the

unbounded dynamic program only described in Section 2.1.1. Both proper and non-proper columns may be generated. However a partial solution can only be augmented with proper columns. Therefore, given the optimal fractional solution $\bar{\lambda}$ of the RMP after column generation, only a variable $\bar{\lambda}_p$ associated with a proper column has to be selected. If such variable exists, the variable is heuristically fixed to an integer value. If there is no such proper variable, another column λ_p is selected. This one has the smallest reduced cost with respect to the optimal dual solution of the RMP among all proper columns contained in the current RMP and proper columns generated by solving the bounded pricing problem with heuristics. The selected column is then added to the partial solution with the value equal to the nearest non-zero integer. Note that if $\bar{\lambda}_p = 0$ then $\lambda_p = 1$ is included in the partial solution.

The 2BP$_l$ has a special structure regarding its subproblems. Standard bin subproblem solutions have a constant contribution to the objective function whereas last bin subproblem solutions are more flexible. Fixing early a column related to a pattern $p \in \mathcal{P}^2$ can have a negative impact on the quality of the final master solution as shown by preliminary experiments. Therefore, the diving heuristic is slightly modified to handle this particularity. Cutting patterns $p \in \mathcal{P}^2$ are never added to the partial solution before patterns $p \in \mathcal{P}^1$. As there is exactly one pattern $p \in \mathcal{P}^2$ in any feasible solution, once it is added to a partial solution, the latter should become complete. Therefore each time the partial solution is augmented with a cutting pattern from $\mathcal{P}^1$, a heuristic starts to check whether the remaining item copies can be cut into one plate of dimension $W \times H$. If it is possible, this produces a cutting pattern $p \in \mathcal{P}^2$ including all remaining item copies and minimizing heuristically its width w_p. Then this pattern is used to complete the solution, and the diving heuristic terminates. This approach of partial solution completion can be extended to be less dependent of the columns fixed during diving. Each time a partial solution is augmented, the whole residual 2BP$_l$ instance is heuristically solved. Calls to these heuristics are done iteratively for each plate until the complete solution of the residual problem is obtained. This modification can be seen as a combination of diving and pricing heuristics.

The diving heuristic is formally represented in Algorithm 6 and starts by calling NON-PROPER-PURE-DIVING$(d, \varnothing, \varnothing, \mathcal{P}^*)$. The pattern set $\mathcal{P}^*$ describes an initial solution to the 2BP$_l$ and can be obtained as described in Section 3.1.3.1. The first step of the algorithm updates the partial fixed solution and the residual demand using previous found pattern p (line 1). After fixing of pattern p, the completion heuristic starts (lines 2-4). A boolean parameter *lastPlateOnly* is used to set how to evaluate the residual 2BP$_l$ instance, building only a solution for the last bin or a complete solution to the residual instance. If this solution improves on the best found one, it is stored. After completion heuristic, the current residual master problem is solved with column generation and unbounded dynamic programming. The obtained frac-

tional solution after convergence is stored in $\bar{\lambda}$. The set of patterns related to proper columns $\mathcal{P}^{prop}$ is then extracted from columns in fractional solution $\bar{\lambda}$ (lines 5-6). If there is at least one proper pattern in $\mathcal{P}^{prop}$, the one of value closest to its nearest non-zero integer value is fixed. This is achieved using operator $\lceil \bar{\lambda}_p \rfloor$ (line 7). In the case of there are no proper columns in the fractional solution $\bar{\lambda}_p$, they should be retrieved elsewhere. Firstly, the set of proper columns $\mathcal{P}^{RMP}$ from the RMP is obtained. Secondly, a set of heuristic patterns $\mathcal{P}^{heur}$ for subproblem $\mathcal{P}^1$ is built. The fixed pattern is then the one of best reduced cost among $\mathcal{P}^{RMP} \cup \mathcal{P}^{heur}$ (lines 8-10). Once a pattern p' to fix has been found, the NON-PROPER-PURE-DIVING recursively calls itself (line 11).

Algorithm 6: NON-PROPER-PURE-DIVING$(d, p, \mathcal{P}^{part}, \mathcal{P}^*)$

1 **if** $p \neq \varnothing$ **then** $d' \leftarrow d - a^p$, $\mathcal{P}^{part} \leftarrow \mathcal{P}^{part} \cup \{p\}$
2 $\mathcal{P}^r \leftarrow$ COMPLETION-HEURISTIC$(d', lastPlateOnly)$
3 **if** $d - \sum_{p' \in \mathcal{P}^r} a^{p'} = \mathbf{0}$ *and* $cost\,(\mathcal{P}^{part} \cup \mathcal{P}^r) < cost(\mathcal{P}^*)$ **then**
4 $\quad \mathcal{P}^* \leftarrow \mathcal{P}^{part} \cup \mathcal{P}^r$
5 solve the RMP with demand bounds d' by column generation and record solution $\bar{\lambda}$
6 $\mathcal{P}^{prop} \leftarrow \{p \in \mathcal{P}^1 :\ \bar{\lambda}_p > 0, a^p \leqslant d'\}$
7 **if** $\mathcal{P}^{prop} \neq \varnothing$ **then** $p' \leftarrow \text{argmin}_{p \in \mathcal{P}^{prop}} \{|\ \bar{\lambda}_p - \lceil \bar{\lambda}_p \rfloor\ |\}$ **else**
8 $\quad \mathcal{P}^{\text{RMP}} \leftarrow$ set of proper patterns $\mathcal{P}^1$ in the RMP
9 $\quad \mathcal{P}^{heur} \leftarrow$ set of heuristic solutions to the pricing problem $\mathcal{P}^1$
10 $\quad p' \leftarrow \text{argmin}_{p \in \mathcal{P}^{\text{RMP}} \cup \mathcal{P}^{heur}} \{\bar{c}_p\}$
11 NON-PROPER-PURE-DIVING$(d', p', \mathcal{P}^{part}, \mathcal{P}^*)$

The "non-proper" diving with Limited Discrepancy Search (LDS) is given in Algorithm 7 and starts by calling NON-PROPER-LDS-DIVING$(d, \varnothing, \varnothing, \mathcal{P}^*, \varnothing, 0)$. First steps of the algorithm are the same as the one in NON-PROPER-PURE-DIVING. The main difference is related to the Limited Discrepancy Search. Instead of fixing a proper column in the fractional solution $\bar{\lambda}$ of the RMP or a heuristic one, a check has to be done regarding the tabu list $\mathcal{Z}$ (lines 3-6). In other words, one needs to ensure that at least one non tabu column is produced by the pricing problem heuristic. This can be achieved by generating a sufficient number of different cutting patterns. The size of set $\mathcal{P}^{heur}$ at line 6 has to be strictly larger than the current size of the tabu list. The NON-PROPER-LDS-DIVING is parametrized with a maximum discrepancy

maxDisc and a maximum depth *maxDepth*.

Algorithm 7: NON-PROPER-LDS-DIVING$(d, p, \mathcal{P}^{part}, \mathcal{P}^*, \mathcal{Z}, depth)$

```
1 execute lines 1-6 from Algorithm 6
2 do
3     if P^prop \ Z ≠ ∅ then p' ← argmin_{p∈P^prop\Z}{| λ̄_p − ⌈λ̄_p⌋ |} else
4         P^RMP ← set of proper patterns P^1 in the RMP
5         P^heur ← set of heuristic solutions to the pricing problem P^1
6         p' ← argmin_{p∈P^RMP ∪ P^heur \ Z}{c̄_p}
7     NON-PROPER-LDS-DIVING(d', p', P^part, P*, Z, depth + 1)
8     Z ← Z ∪ {p'}
9 while |Z| ⩽ maxDisc and depth ⩽ maxDepth
```

Note that if the 2BP has to be solved, the only modification to perform on Algorithms 6 and 7 is to change how patterns are created with the completion heuristic.

3.1.2.2 Completion heuristic

The completion heuristic is a key point in Algorithms 6 and 7. Its aim is to provide a set of valid cutting patterns such that combining this set with a partial solution gives a valid solution for the $2BP_l$. This heuristic is a modified version of a list heuristic for the 2BP and uses heuristics for the 2KP.

The general framework of the completion heuristic is to take iteratively each bin, build a valid pattern for it and repeat those steps while there is at least an item to cut. An overview of the procedure is given in Algorithm 8. At each iteration, the hypergraph corresponding to the bin is selected. Then two valid patterns are created using the evolutionary algorithm and the hypergraph constructive heuristic, mentioned in Section 2.5. Each heuristic returns one feasible pattern. The best one among the two is then kept (lines 4-6). The residual demand, area of item to cut and found pattern set are then updated (lines 6-7). When all items are cut, all found patterns are marked as a valid solution for the subproblem $\mathcal{P}^1$. The last found pattern is however marked as a valid solution for the subproblem $\mathcal{P}^2$ (lines 10-11). From a practical point of view, the only difference between creating a pattern related to $\mathcal{P}^1$ or $\mathcal{P}^2$ is based on the way to evaluate its value. In the first case, its value is just the total area of cut items in it. In the second case, its value is the total area of cut items but also takes into account the total cut length. Clearly, one can convert a pattern related to $\mathcal{P}^1$ in a pattern for $\mathcal{P}^2$. This is why the two last steps of the algorithm mark all patterns as belonging to $\mathcal{P}^1$ and then only the last created one to $\mathcal{P}^2$. The behaviour of the completion heuristic depends on the input boolean parameter *lastPlateOnly*. When it is set to true, only one pattern for the last bin subproblem is created and the method exits returning

only this pattern.

Algorithm 8: COMPLETION-HEURISTIC($d, lastPlateOnly$)

$\mathcal{P}^r \leftarrow \emptyset$, $A \leftarrow \sum_{i \in \mathcal{I}} w_i h_i d_i$
$G \leftarrow$ hypergraph related to $\mathcal{P}^1$
do
  if $A \leqslant W \times H$ *or* $lastPlateOnly$ **then** $G \leftarrow$ hypergraph related to $\mathcal{P}^2$ $p \leftarrow$ pattern from evolutionary algorithm on bin (W, H) with demand d
  $p' \leftarrow$ pattern using hypergraph heuristic on G
  if $val(p') > val(p)$ **then** $p \leftarrow p'$ $d \leftarrow d - a^p$, $A \leftarrow A - \sum_{i \in \mathcal{I}} w_i h_i a_i^p$
  $\mathcal{P}^r \leftarrow \mathcal{P}^r \cup \{p\}$
  if $lastPlateOnly$ **then** break
while $d \neq 0$
mark all patterns in $\mathcal{P}^r$ as related to $\mathcal{P}^1$
mark the last inserted pattern in $\mathcal{P}^r$ as related to $\mathcal{P}^2$
return $\mathcal{P}^r$

3.1.3 Initial bounds

To avoid starting diving heuristic and hypergraph building for easy instances, initial primal and duals bounds can be first computed for the 2BP$_l$. If one finds a primal bound equal to a dual bound, one can state that optimality is reached and stop immediately. If not then one can use the primal solution to initialize columns for the 2BP$_l$. A good primal bound also helps the diving heuristic by avoiding the exploration of non promising branches in the diving tree. This section describes both primal and dual bounds for the 2BP$_l$ but also valid for the 2BP.

3.1.3.1 Initial primal bounds

A direct way to obtain a primal solution for the 2BP$_l$ is to use the evolutionary algorithm described in Section 2.5.2 iteratively for each bin until there is no more items to cut. This heuristic is called (*ea*).

Alternatively one can use standard bin-packing list heuristics described in Section 1.4.3. These heuristics run in polynomial time and produce a feasible packing for the 2BP$_l$. To exploit the fast running time of list heuristics, they are used as a subroutine in a metaheuristic called (*iub*). This one combines both list heuristics and the evolutionary algorithm. Let consider $10*|\mathcal{I}|$ random permutations of set $\bar{\mathcal{I}}$. List heuristics mentioned in Section 1.4.3 are applied for each of these permutations. These heuristics can be used in several ways. The first option is to fill one bin at a time, the next bin is opened when there are no items in the current list that fit into the current bin. The second option is to open all bins at the same time and fill them. A third option is to

implement the two-phase heuristics described in Section 1.4.3.2. At the same time, an extra bin-packing solution is obtained by solving iteratively 2KPs with the evolutionary algorithm. For the latter, 10 initial random populations are generated and each of them is improved during $|\mathcal{I}|/10$ iterations.

Using the three described options as well as the evolutionary algorithm, a set of feasible bin-packing solutions for each random permutation of set $\bar{\mathcal{I}}$ is obtained. The best solution providing a valid primal bound for the 2BP$_l$ is first recorded. Then, among all bin-packing solutions, the bin of smallest waste is selected and added to a partial bin-packing solution. The whole process is then reiterated on the residual problem. The algorithm terminates when there are no more items to cut.

3.1.3.2 Initial dual bounds

To have an estimation of the quality of an initial primal bound, bin-packing dual bounds are computed. If the gap between dual and primal bounds is small, this gives a good assessment of the value of an optimal solution.

A trivial dual bound for the 2BP$_l$ is obtained in the same fashion as the 2BP. Since all bins are identical in a 2BP$_l$ instance, a trivial dual bound on the total length of a solution is given by:

$$DB_1 = W \times \left\lceil \frac{\sum_{i \in \mathcal{I}} w_i h_i d_i}{W \times H} \right\rceil \tag{3.13}$$

This bound does not take into account the used length of the last bin. It can be strengthen by computing a dual bound as done for the 2SP:

$$DB_2 = \left\lceil \frac{\sum_{i \in \mathcal{I}} w_i h_i d_i}{H} \right\rceil \tag{3.14}$$

This bound gives an approximation of the value of an optimal solution. Clearly these dual bounds are dominated by the one obtained using column generation.

3.1.4 Computational experiments

This section reports results for the different ways to solve the 2BP and 2BP$_l$. Datasets used for experiments are the ones from the literature, named $C1-10$. They were introduced in Berkey and Wang [7] and Martello and Vigo [55]. More details can be found in Section 1.4.1. To ease reading, instances in datasets $C1-10$ are grouped by number of items $|\mathcal{I}|$. The notation $C-I20$ refers to all instances containing 20 items among datasets $C1-10$. Experiments also consider real-world instances. Their configuration is the same as

the one detailed in Chapter 2. A reminder is given here. The bin size is set to $(3000, 1500)$ and $(6000, 3000)$. The number of different items $|\mathcal{I}|$ in an instance can be 50, 100 and 150. The average demand of an item is two. When the bin size is $(6000, 3000)$, the average maximum item size is around $(2000, 1100)$, its average minimum size is around $(500, 200)$. For bin size equal to $(3000, 1500)$, item maximum and minimum dimensions are divided by half. A dataset named $B6000I50$ corresponds an instance with $B = (6000, 3000)$ and $|\mathcal{I}| = 50$. Respectively, a dataset named $B3000I150$ corresponds an instance with $B = (3000, 1500)$ and $|\mathcal{I}| = 150$. Each dataset contains 50 instances, this gives a total of 300 instances. Datasets are grouped by their number of items $|\mathcal{I}|$. The notation $B - I50$ refers to all instances containing 50 items in datasets $B3000I50$ and $B6000I150$.

The goal of experiments for the 2BP and 2BP$_l$ is twofold: (i) to evaluate the impact of the pricing problem with partial enumeration on lower and upper bounds produced with the column generation; (ii) to evaluate the quality of the solutions produced by different variants of the diving heuristic and compare them to heuristics described in Section 3.1.3.1.

All experiments are run using a 2.5 Ghz Haswell Intel Xeon E5-2680 with 128Go of RAM. CPLEX 12.6 is used to solve linear programs. The time limit to solve one instance is set to one hour.

3.1.4.1 Impact of the partial enumeration

From the hypergraph representation defined in Chapter 2, it has been outlined that using different configurations to build it gives different results regarding the efficiency of exact methods. In practice, the partial pattern enumeration strengthens the item production bounds with modification of the hypergraph structure. The goal of this section is twofold. The first one is to check the impact of the partial pattern enumeration when solving pricing problems in a column generation context for 2BP and 2BP$_l$. The second is to validate that solving the unbounded pricing problem instead of the bounded one does not degrade the quality of the master linear relaxation after column generation convergence.

Remember that the configuration of the partial enumeration is characterized by four values: Δ_w^{size}, Δ_h^{size}, Δ_w^{diff}, and Δ_h^{diff}. Three ways configurations are retained. The first one corresponds to the hypergraph building using all basic simplifications and no pattern enumeration: $\Delta_1 = (\Delta_w^{size} = 0, \Delta_h^{size} = 0, \Delta_w^{\text{diff}} = 0, \Delta_h^{\text{diff}} = 0)$. The second one corresponds to the hypergraph building with enumeration of items with the same h_i for odd cutting stages and with the same w_i for even cutting stages: $\Delta_2 = (\Delta_w^{size} = 1000, \Delta_h^{size} = 1000, \Delta_w^{\text{diff}} = 1, \Delta_h^{\text{diff}} = 1)$. The last setting corresponds to the hypergraph building with enumeration of items with the same h_i for odd cutting stages and enumeration of items with different w_i at the second cutting

stage: $\Delta_3 = (\Delta_w^{size} = 1000, \Delta_h^{size} = 1000, \Delta_w^{\text{diff}} = \infty, \Delta_h^{\text{diff}} = 1)$. The retained configurations are the one already described in Section 2.6.2.

Results are reported in Tables 3.1 for the 2BP and in Table 3.2 for the $2BP_l$. In both tables, the first column reports the instance class and its total number of instances between brackets. The column *#pp* shows the number of instances solved to optimality in preprocessing using the fast heuristic (*ea*) from Section 3.1.3.1. Optimality is proved if the obtained primal solution value is equal to the value of a trivial dual bound from Section 3.1.3.2. The next three columns present results for the variant cg_{mip}. For this variant, pricing problems are solved with MILP using hypergraph and configuration Δ_1. The reported values are the number of instances not solved by (*ea*) for which the column generation converged within one hour, the average time t and the average primal-dual gap *gap* in percentage from a best known solution. Next three columns give the average gap $gap_{\#opt}$ for other three variants (Δ_1, Δ_2, Δ_3) of column generation using unbounded dynamic program as pricing oracle. In order to have a correct comparison, averages in columns $gap_{\#opt}$ are calculated only for instances for which the variant cg_{mip} converged. Note that the column generation variants with dynamic programming as pricing oracle converged within the time limit for all instances. Thus in columns gap_{all} and t_{all}, the average gap and the average computation time are computed among all instances not solved optimally by heuristic (*ea*) for all hypergraph configurations. Column generation is initialized with columns from primal solution. Reported times are in seconds and round-up. In Table 3.2 column (*#pp*) is omitted since no instances are solved in preprocessing.

		cg_{mip}			$gap_{\#opt}$,%			gap_{all}, %			t_{all}		
Instances	*#pp*	*#opt*	t	*gap*, %	cg_{Δ_1}	cg_{Δ_2}	cg_{Δ_3}	cg_{Δ_1}	cg_{Δ_2}	cg_{Δ_3}	cg_{Δ_1}	cg_{Δ_2}	cg_{Δ_3}
C-I20 (100)	63	37	71	6.9	8.7	8.4	8.3	8.7	8.4	8.3	0.2	0.1	0.1
C-I40 (100)	37	52	460	2.6	3.1	2.9	2.9	6.0	5.8	5.8	0.6	0.6	0.6
C-I60 (100)	35	32	853	0.9	1.3	1.1	1.1	3.3	3.1	3.1	1.4	1.3	1.3
C-I80 (100)	29	24	760	0.7	1.0	0.8	0.8	2.9	2.8	2.8	2.7	2.6	2.5
C-I100 (100)	31	20	705	0.5	0.7	0.6	0.6	2.9	2.8	2.8	4.8	4.8	4.7
B-I50 (100)	76	0	-	-	-	-	-	13.8	13.7	13.7	1.8	1.7	1.8
B-I100 (100)	61	0	-	-	-	-	-	6.9	6.9	6.9	12.3	12.0	12.5
B-I150 (100)	50	0	-	-	-	-	-	4.7	4.7	4.7	52.8	47.7	51.3

Table 3.1: Comparison of different column generation variants for the 2BP

From Table 3.1, a direct observation is that the dynamic program is orders of magnitude faster than MIP for solving the pricing problem. The partial enumeration increases the running time of column generation only marginally. In the same time, performing an enumeration with configuration Δ_3 allows one to obtain a lower bound which is close to the "proper" lower bound, at least for the easiest instances that can be tackled by cg_{mip}. The "proper" bound for

	cg_{mip}			$gap_{\#opt}$,%			gap_{all}, %			t_{all}		
Instances	$\#opt$	t	gap, %	cg_{Δ_1}	cg_{Δ_2}	cg_{Δ_3}	cg_{Δ_1}	cg_{Δ_2}	cg_{Δ_3}	cg_{Δ_1}	cg_{Δ_2}	cg_{Δ_3}
C-I20 (100)	62	239	2.0	3.9	3.2	3.1	3.8	3.1	3.0	0.3	0.4	0.3
C-I40 (100)	58	1391	1.2	1.8	1.5	1.5	2.7	2.4	2.4	1.6	1.6	1.6
C-I60 (100)	25	881	0.6	0.9	0.7	0.7	1.8	1.6	1.6	3.6	3.8	3.7
C-I80 (100)	19	616	0.6	0.8	0.6	0.6	1.4	1.3	1.3	6.4	6.6	6.4
C-I100 (100)	15	562	0.5	0.6	0.6	0.6	1.3	1.1	1.1	10.6	11.4	10.9
B-I50 (100)	0	-	-	-	-	-	1.4	1.2	1.2	6.8	6.3	6.7
B-I100 (100)	0	-	-	-	-	-	0.7	0.7	0.7	43.0	37.2	40.6
B-I150 (100)	0	-	-	-	-	-	0.6	0.5	0.5	129.6	109.6	116.7

Table 3.2: Comparison of different column generation variants for the 2BP$_l$

larger instances is not obtained within the time limit. Consequently, it is not possible to measure how close is the lower bound obtained by $cg_{dp\Delta_3}$ to the "proper" one. Note that at least a third of the instances in each dataset are solved to optimality in preprocessing for literature instances, it is a half for real-world dataset. Remark also that the shown gap is high in practice. This is related to the flat objective function of the 2BP.

From Table 3.2, the same observations about dynamic program is done. It is still faster than using MIP solver for pricing problems. More advanced pattern enumerations also lead to a lower bound which is close to the "proper" lower bound.

Application of the partial enumeration technique significantly increases the quality of lower bounds obtained by column generation at virtually no cost. Therefore, it offers a good trade-off between the quality of lower bounds and the column generation running time. It can also be seen that column generation is slower for real-life instances. This is logical as the running time of the dynamic program depends on the plate size, which is larger for the instances in datasets B.

3.1.4.2 Comparison of exact methods and heuristics

In this section, an estimation of the impact of the partial enumeration on the pseudo-polynomial formulation (3.7)–(3.12) is done. From results of Martin et al. [56], in the absence of constraints (3.10), this formulation has the integrality property. Thus the value of its linear programming relaxation is equal to the lower bound obtained by column generation with "non-proper" columns. As outlined in Section 3.1.4.1, the latter may be increased using partial enumeration. Thus, the strength of the linear programming relaxation of formulation (3.7)–(3.12) based on partly enumerated hypergraph may be improved too. Moreover, variables x may have coefficients greater than one if there are hyperarcs which correspond to cutting several copies of one item,

as in the case of cutting a meta-item. Thus, partial enumeration enables the possibility of adding knapsack cutting planes when solving the formulation by a MIP solver.

The results for direct solution by the CPLEX MIP solver of formulation (3.7)–(3.12) with different settings of the partial enumeration of hypergraph are shown in Tables 3.3 and 3.4. The first reported value is the number of instances solved in preprocessing *#pp*. Then for each hypergraph configuration, the number of remaining instances solved in one hour *#opt* with MIP formulation is reported. In the last part of the table, the average time t required to solve all instances is reported. If an instance is not solved within the time limit, the time limit value is used in the calculation of the average time. Reported times are in seconds and round-up. In Table 3.4 column (*#pp*) is omitted since no instances are solved in preprocessing.

		#opt			t		
Instances (#)	*#pp*	$MIP\Delta_1$	$MIP\Delta_2$	$MIP\Delta_3$	$MIP\Delta_1$	$MIP\Delta_2$	$MIP\Delta_3$
C-I20 (100)	63	37	37	37	0.6	0.4	0.4
C-I40 (100)	37	60	61	61	126.7	87.9	92.3
C-I60 (100)	35	60	62	62	283.8	164.8	171.7
C-I80 (100)	29	60	62	62	514.8	446.6	457.3
C-I100 (100)	31	56	57	56	747.2	646.5	646.3
B-I50 (100)	76	3	4	4	807.3	777.8	781.3
B-I100 (100)	61	0	0	0	1408	1407.9	1407.9
B-I150 (100)	50	0	0	0	1810.3	1809.8	1809.9

Table 3.3: Comparison of hypergraph-based MIP formulations for the 2BP with different partial enumeration

	#opt			t		
Instances (#)	$MIP\Delta_1$	$MIP\Delta_2$	$MIP\Delta_3$	$MIP\Delta_1$	$MIP\Delta_2$	$MIP\Delta_3$
C-I20 (100)	91	98	97	434.8	237.2	253.7
C-I40 (100)	72	74	75	1238.6	1220.9	1196.6
C-I60 (100)	53	59	57	1862.9	1827.1	1823.7
C-I80 (100)	48	47	47	2050.4	2059.2	2063.4
C-I100 (100)	40	42	41	2287.8	2212.7	2234
B-I50 (100)	0	0	0	3600	3600	3600
B-I100 (100)	0	0	0	3600	3600	3600
B-I150 (100)	0	0	0	3600	3600	3600

Table 3.4: Comparison of hypergraph-based MIP formulations for the $2BP_l$ with different partial enumeration

According to the results in Table 3.3, 2BP can be solved to optimality with hypergraph based MIP formulations. When more patterns are enumerated

in the hypergraph, the average computation time decreases. Nevertheless, it does not seem to be the best method to find solution for B datasets. Indeed among instances not solved in preprocessing, almost none of them are solved with MIP formulations. The same conclusions are made from results reported in Table 3.4. The pseudo-polynomial size ILP formulation using hypergraph is a possible way to solve both 2BP and 2BP$_l$. When more patterns are enumerated, the average computation time decreases and more instances are solved. However, the improvement is not radical. One can also notice that, when the 2BP and the 2BP$_l$ are solved, B datasets are significantly harder to solve. As a general observation, the 2BP$_l$ is much harder to solve compare to the 2BP.

Since exact methods do not guarantee to solve all problem instances in a reasonable amount of time, heuristics are preferred. Results using some of them are discussed hereinafter. In this experiments, five heuristics are used:

- evolutionary algorithm (ea);
- algorithm (iub), which combines evolutionary algorithm with list heuristics;
- a diving heuristic denoted ($div_{\varnothing}$) without partial enumeration in the pricing problem and with simple evaluation of the residual problem in the diving (parameter $lastPlateOnly = true$);
- a diving heuristic denoted (div) with partial enumeration Δ_3 in the pricing problem and with simple evaluation of the residual problem in the diving (parameter $lastPlateOnly = true$);
- a combination of the diving heuristic and the evolutionary algorithm with complete evaluation of the residual problem in the diving (parameter $lastPlateOnly = false$), called ($ediv$).

The first two heuristics are presented in Section 3.1.3.1. The other three heuristics are described in Section 3.1.2. Variants with Limited Discrepancy Search for the last two heuristics and denoted as (div_{32}) and ($ediv_{32}$) are also used. In these variants, the backtrack is allowed up to the depth of 2 of the search tree and the maximum size of the tabu list is 3. This LDS parametrisation results in at most ten dives in the search tree. Diving heuristics are initialized with the solution produced by heuristic (ea).

In Tables 3.5 and 3.6, the average gap gap in percentage from a best known solution and the average time t for all heuristics and their variants are reported. Times are in seconds and round-up.

For the 2BP and from Table 3.5, the initial heuristic (ea) provides good results in a short computation time. The combination of (ea) with list heuristics obtains better results but require more computation time. From a global point

	ea		*iub*		$div_{\varnothing}$		*div*		div_{32}		*ediv*		$ediv_{32}$	
Instances	*gap*	*t*	*gap*	*t*	*gap*	*t*	*gap*	*t*	*gap*	*t*	*gap*	*t*	*gap*	*t*
C-I20 (100)	1.13	1	0.20	1	0.82	1	0.78	1	0.00	1	0.78	1	0.00	1
C-I40 (100)	2.63	1	1.24	3	1.80	1	1.94	1	1.27	1	1.94	1	1.27	1
C-I60 (100)	3.47	2	1.77	8	1.86	2	2.02	2	1.06	2	2.02	2	1.06	2
C-I80 (100)	3.76	3	1.29	21	1.28	3	1.14	3	0.28	4	1.14	3	0.28	4
C-I100 (100)	3.79	4	1.25	39	0.75	5	0.79	6	0.45	7	0.79	5	0.45	7
B-I50 (100)	0.08	2	0.08	3	0.08	2	0.08	2	0.08	3	0.08	2	0.08	2
B-I100 (100)	0.04	7	0.00	24	0.04	12	0.04	13	0.00	23	0.04	13	0.00	23
B-I150 (100)	0.06	21	0.00	128	0.06	47	0.03	50	0.03	83	0.03	48	0.03	84

Table 3.5: Comparison of heuristics for the 2BP

	ea		*iub*		$div_{\varnothing}$		*div*		div_{32}		*ediv*		$ediv_{32}$	
Instances	*gap*	*t*	*gap*	*t*	*gap*	*t*	*gap*	*t*	*gap*	*t*	*gap*	*t*	*gap*	*t*
C-I20 (100)	3.47	1	2.11	1	2.13	1	2.20	1	1.68	1	1.88	1	1.61	1
C-I40 (100)	3.65	1	1.80	4	1.43	2	1.12	2	0.53	5	0.84	3	0.38	10
C-I60 (100)	3.68	2	1.80	13	1.22	4	0.95	4	0.31	10	0.65	9	0.19	30
C-I80 (100)	4.18	3	1.97	32	1.00	6	0.66	6	0.17	20	0.33	19	0.07	77
C-I100 (100)	4.47	4	1.84	64	0.57	10	0.60	10	0.13	34	0.32	38	0.05	178
B-I50 (100)	2.35	2	2.24	16	1.10	12	0.63	13	0.12	63	0.32	17	0.00	95
B-I100 (100)	1.90	7	1.80	179	0.61	99	0.34	104	0.09	662	0.18	154	0.00	1077
B-I150 (100)	1.54	21	1.46	750	0.43	342	0.15	347	-	3600	0.07	581	-	3600

Table 3.6: Comparison of heuristics for the $2BP_l$

of view, diving heuristics outperform constructive ones. When looking closer, heuristics ($div_{\varnothing}$) and (*div*) are close to each other both in term of computation time and gap. It seems that using the diving heuristic with total completion heuristic (*ediv*) is not improving compare to (*div*). Moreover, using diving heuristics with LDS achieve the best results. For literature datasets, computation time of all diving heuristics are pretty much the same, so the diving with LDS is the best method to keep. Nevertheless, for B datasets, gaps are close to each other for all methods. It seems that their is no real gain to obtain to select diving with LDS in that case. Solution quality from initial heuristic (*ea*) is enough in that case.

On the opposite from Table 3.6 for the $2BP_l$, the diving heuristic is clearly the method to choose. Indeed the initial heuristic (*ea*) is the fastest method but produce solutions of worst quality. Its variant (*iub*) improves the solution quality at the expense of much larger running time. However, it struggles with B datasets, as the solution improvement over (*ea*) is very small. Diving algorithms ($div_{\varnothing}$) and (*div*) significantly outperform heuristic (*iub*) both in terms of running time and solution quality. The partial enumeration tech-

nique increases the effectiveness of the diving heuristic for most instances at a very small cost. This technique is especially useful for large instances. The combination (*ediv*) of the diving heuristic and the evolutionary algorithm further improves the quality of the obtained solutions at a cost of a reasonable increase of running time except for instance with $|\mathcal{I}| = 150$. The best solutions on average are obtained by diving heuristics with Limited Discrepancy Search. However the running time of these heuristics is quite long especially for large instances. It is also not possible to solve within one hour the instance with the largest number of items. Thus, the heuristics (*div*) and (*ediv*) offer the best tradeoff between solution quality and running time.

3.2 Solving consecutive 2BP and 2BP$_l$

The subject of this chapter is to solve the 2BP$_l$ for a given item batch. A more global problem and also to benchmark 2BP$_l$ solving methods is to consider the problem in its consecutive variant. This variant is related to the factory organization where different item batches are cut one after each other during the day. This is the main topic of this section. The problem is first described and its formulations given. Then the "non-proper" diving heuristic is adapted for this problem and results are outlined on real-world instances.

3.2.1 Problem description

The consecutive 2BP$_l$ has two uncommon specificities. The first is that production is decomposed into item batches. This is due to the fact that there is a limited intermediate storage area between the cutting and assemblage production units. Thus the set of glass pieces to produce during a day is predecomposed into batches such that each batch fits to the storage. Batches are cut in a predetermined specific order that takes into account the due dates of customer orders. In a given batch, the exact specified quantity of ordered glass pieces has to be cut. It is forbidden to have overproduction or underproduction. The number of plates is always sufficient to cut all ordered pieces. The second specificity is the way leftovers are handled. There is no specific area to store leftovers from previous cutting patterns, mostly because there is no standard size for the orders, and organization costs that would be entailed are expected to be larger than the cost of the raw material saved by reusing all leftovers. Therefore, almost all leftovers are recycled, and cannot be used for subsequent batches. Only one leftover piece is kept from each batch: the one related to the last plate used. This subplate remains on the cutting device. Its height must be equal to the height of the large plates. Figure 3.1 depicts feasible/infeasible solutions.

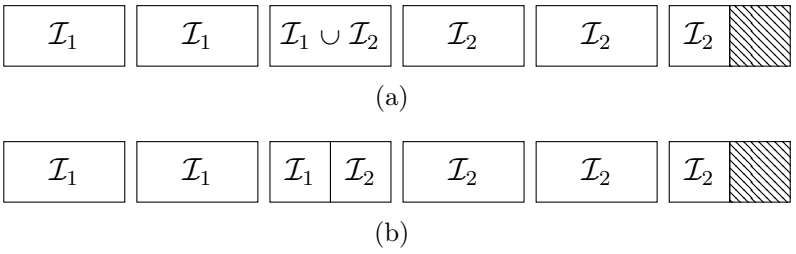

Figure 3.1: Representation of a solution for two batches $\mathcal{I}_1$ and $\mathcal{I}_2$. The first solution (a) is infeasible since it is not allowed to mix items from different batches in a cutting pattern. The second solution (b) is feasible. The leftover from batch $\mathcal{I}_1$ is used to cut items from batch $\mathcal{I}_2$.

Formally a consecutive 2BP (C-2BP) is a modified version of the 2BP. A problem instance is represented by a pair $D = (\mathbf{I}, B)$, where $\mathbf{I} = \{\mathcal{I}_1, \ldots, \mathcal{I}_n\}$ is an ordered set of batches to cut and B is the standard bin to use of fixed width and height (W, H). Bins are assumed to be identical and in a quantity large enough to cut all items in $\mathbf{I}$. Each item i has a fixed width and height (w_i, h_i) and has to be cut exactly d_i times, Each batch is cut in the order in which it occurs in $\mathbf{I}$. The problem objective is to minimize the number of bins to use in order to cut all batches. Since to cut each item batch, the problem is to minimize the number of bins, solving an instance of C-2BP is equivalent to solve one 2BP for each batch in $\mathbf{I}$.

The consecutive 2BP$_l$ (C-2BP$_l$) is an extension of C-2BP. A problem instance is still represented by a pair $D = (\mathbf{I}, B)$, where $\mathbf{I} = \{\mathcal{I}_1, \ldots, \mathcal{I}_n\}$ is an ordered set of batches to cut and B is the standard bin to use of fixed width and height (W, H). Bins are assumed identical and in a quantity large enough to cut all items in $\mathbf{I}$. Each item i has a fixed width and height (w_i, h_i) and has to be cut exactly d_i times, Each item batch has to be cut in the order in which it occurs in $\mathbf{I}$. Problem objective is to minimize the number of bins to use in order to cut all batches. To reduce the loss of material, it is allowed to use the leftover from the last cut bin of the previous batch to initialize the current batch. Assume a batch $\mathcal{I}_s, s > 2$ to process. Possible bins to use are the residual bin from batch $s-1$ of given size $\tilde{W} \times H$ and an unlimited number of standard bins of size $W \times H$. Since leftovers are considered, the total used width in the last bin has to be minimized. Contrary to C-2BP, obtained solutions for each batch for the C-2BP$_l$ are linked together. Indeed the solution related to a given batch $\mathcal{I}_s, s > 2$ is dependant of the solution obtained for batch $\mathcal{I}_{s-1}$.

The main difference between C-2BP and C-2BP$_l$ is the possibility to use residual plates from the previous batch. From a practical point of view, discarding residual plates in the C-2BP is interesting since it requires less handling from the operator. It clearly implies to throw away a large quantity of raw material. Considering residual plates in the C-2BP$_l$ requires more handling

from the operator but may lead to save more raw material.

Such problem is not very common but an application has been proposed in Birgin et al. [8]. Authors consider a non-guillotine multiperiod 2BP$_l$ problem where items have to be cut at different time period and with varying costs from one period to another. Leftovers from previous period are available to cut items in current and next periods. The described problem aims to minimize the overall cost of cut items and maximizing the value of usable leftovers at the last time period. A parallel can be drawn with the C-2BP$_l$. Firstly the notion of period in problem described in Birgin et al. [8] is similar to the notion of item batches. Batches are cut one after each other and thus this corresponds to a period. Secondly it is possible to use leftovers between periods. Nevertheless there exist some differences. Indeed in the C-2BP$_l$, each item among all batches has no varying cost. The main of objective for each period is to find feasible cutting patterns to cut all items in considered batch. Moreover to avoid a too important storage of leftovers and to reduce handling from operators, the C-2BP$_l$ only considers the last residual plate from the previous period as usable to cut batch for the current period.

3.2.2 Problem formulations

From the definition of C-2BP and C-2BP$_l$, direct ILPs can be written using the existing ones for the 2BP and 2BP$_l$. Since no residual bin parts are carried from one batch to another in C-2BP, this leads to solve an independent 2BP for each batch in $\mathbf{I}$. The optimum of C-2BP is then achieved by finding the optimal solution of each 2BPs. A straightforward ILP model for the C-2BP is to write $|\mathbf{I}|$ 2BP formulations, one for each batch $\mathcal{I} \in \mathbf{I}$. Thus, the formulation of the C-2BP is not reported since it just a simple rewriting.

On the contrary, the C-2BP$_l$ has to handle residual bin from one batch to another. The data for the 2BP$_l$ for a batch $\mathcal{I}_s, s > 2$ depends therefore on the residual bin obtained after cutting batch $\mathcal{I}_{s-1}$. The optimal solution of the C-2BP$_l$ is obtained by solving consecutively 2BP$_l$, one for each batch $\mathcal{I} \in \mathbf{I}$. A slight modification occurs on the 2BP$_l$ formulation however. Since residual bin part is used for a given batch, the ILP formulation of the 2BP$_l$ has to be enhanced to handle this case. This expresses by adding a new subproblem to the 2BP$_l$ ILP model (3.1)-(3.5). Let $\mathcal{P}^0$ be the set of valid patterns related to the residual bin of width $\tilde{W}$ obtained from the cut of a previous batch. The ILP model of the nested 2BP$_l$ for a given batch $\mathcal{I}_s, s > 2$ in order to solve the C-2BP$_l$ is the following:

$$\min \tilde{W} \sum_{p \in \mathcal{P}^0} \lambda_p + \sum_{p \in \mathcal{P}^1} W \lambda_p + \sum_{p \in \mathcal{P}^2} w_p \lambda_p \tag{3.15}$$

$$\text{s.t.} \sum_{p \in \mathcal{P}^0} a_{ip} \lambda_p + \sum_{p \in \mathcal{P}^1} a_{ip} \lambda_p + \sum_{p \in \mathcal{P}^2} a_{ip} \lambda_p = d_i, \quad \forall i \in \mathcal{I}_s \tag{3.16}$$

$$\sum_{p\in\mathcal{P}^0} \lambda_p = 1 \tag{3.17}$$

$$\sum_{p\in\mathcal{P}^2} \lambda_p = 1 \tag{3.18}$$

$$\lambda_p \in \{0,1\}, \quad \forall p \in \mathcal{P}^0 \tag{3.19}$$

$$\lambda_p \in \mathbb{N}, \quad \forall p \in \mathcal{P}^1 \tag{3.20}$$

$$\lambda_p \in \{0,1\}, \quad \forall p \in \mathcal{P}^2 \tag{3.21}$$

Objective function (3.15) minimizes the total used length to cut batch $\mathcal{I}_s$. Constraints (3.16) ensure to cut all items among residual bin, standard bins and the last bin. Convexity constraint (3.17) (resp. (3.18)) imposes to apply one cutting pattern exactly to the residual bin (resp. the last bin).

The ILP formulation (3.15)-(3.21) is very close to the one defined by (3.1)-(3.5). When the first batch $\mathcal{I}_1 \in \mathbf{I}$ is cut, formulations are equivalent since there is no residual from previous batch. The main difference is related to the new pricing problem related to pattern $p \in \mathcal{P}^0$ for a bin of width $\tilde{W}$. Let κ be the dual variable related to (3.17), the reduced cost of the pricing problem is:

$$\zeta(\lambda_p) = \tilde{W} - \sum_{i\in\mathcal{I}} a_{ip}\pi_i - \kappa \tag{3.22}$$

Finding a pattern $p \in \mathcal{P}^0$ is equivalent to solve a C-2KP-RE-4-r with an input bin $(\tilde{W}, H)$.

It is also possible to obtain a pseudo-polynomial size formulation for the C-2BP$_l$ derived from the one outlined in Section 3.1.1.2. In that case, one needs to index a new hypergraph by $k = 1$ related to the residual bin $(\tilde{W}, H)$. Using the notations used for (3.7)-(3.12), the extended pseudo-polynomial size ILP formulation for the C-2BP$_l$ is given by:

$$\min \tilde{W}z_1 + Wz_2 + z_3 \sum_{a\in\bar{\mathcal{A}}_3} w_a x_a \tag{3.23}$$

$$\text{s.t.} \sum_{a\in\Gamma^-(v)} x_a - \sum_{a'\in\Gamma^+(v)} n_{a'}(v)x_{a'} = 0, \quad \forall v \in \mathcal{V}_k\backslash\{t_k \cup \mathcal{I} \cup \varnothing\}, \forall k \in \{1,2,3\} \tag{3.24}$$

$$\sum_{a\in\Gamma^-(t_k)} x_a = z_k, \quad \forall k \in \{1,2,3\} \tag{3.25}$$

$$\sum_{k\in\{2,3\}} \sum_{a\in\mathcal{A}_k(i)} n_a(i)x_a = d_i, \quad \forall i \in \mathcal{I} \tag{3.26}$$

$$z_1 = z_3 = 1, \quad z_2 \in \mathbb{N} \tag{3.27}$$

$$x_a \in \mathbb{N}, \quad \forall a \in \mathcal{A}_k, \forall k \in \{1,2,3\} \tag{3.28}$$

Objective function (3.23) minimizes the number of patterns for the residual bin, standard bins and the length of the pattern selected for the last bin. The contribution to the objective function is given by the sum of the width w_a of hyperarcs $a \in \bar{\mathcal{A}}_3$ where $\bar{\mathcal{A}}_3$ is the set of hyperarcs associated to cuts performed only at the first cutting stage. Note that only one pattern can be created for the residual and the last bin subproblem according to constraint (3.27) and problem definition. Constraints (3.8)-(3.9) are classical flow conservation constraints. Constraint set (3.10) ensures that the right amount of item $i \in \mathcal{I}$ are cut among all patterns.

3.2.3 Modified diving heuristic

When solving the pure 2BP$_l$ by column generation, there are two pricing subproblems. The first is related for standard bins and the second for the last bin to minimize its used length. When the 2BP$_l$ is solved as subroutine for the C-2BP$_l$, one needs to consider the third subproblem related to the residual bin $(\tilde{W}, H)$ of the previous batch. A way to tackle the C-2BP$_l$ is to solve one 2BP$_l$ instance for each batch in $\mathbf{I}$. Therefore one just needs to embed the diving procedure for the 2BP$_l$ in an iterative method. The diving heuristic introduced in Section 3.1.2 is slightly modified for that case. The modified diving heuristic, given in Algorithm 6, is then used as a subroutine to derive a valid solution for the C-2BP$_l$. In the context of solving the C-2BP$_l$, this algorithm has to be modified to handle the pricing subproblem related to residual plate $\mathcal{P}^0$. This is formally represented in Algorithm 9 which contains simple modifications in comparisons to Algorithm 6. Indeed, sets of patterns $\mathcal{P}^{prop}$ and $\mathcal{P}^{\text{RMP}}$ contain now both patterns from $\mathcal{P}^1$ and $\mathcal{P}^0$.

Algorithm 9: NON-PROPER-PURE-DIVING$_m(d, p, \mathcal{P}^{part}, \mathcal{P}^*)$

1 call lines 1-4 from Algorithm 6
2 solve the RMP with demand bounds d' by column generation and record solution $\bar{\lambda}$
3 $\mathcal{P}^{prop} \leftarrow \{p \in \mathcal{P}^0 \cup \mathcal{P}^1 : \bar{\lambda}_p > 0, a^p \leqslant d'\}$
4 **if** $\mathcal{P}^{prop} \neq \varnothing$ **then** $p' \leftarrow \text{argmin}_{p \in \mathcal{P}^{prop}}\{|\ \bar{\lambda}_p - \lceil \bar{\lambda}_p \rfloor\ |\}$ **else**
5 $\quad \mathcal{P}^{\text{RMP}} \leftarrow$ set of proper patterns $\mathcal{P}^0 \cup \mathcal{P}^1$ in the RMP
6 $\quad \mathcal{P}^{heur} \leftarrow$ set of heuristic solutions to the pricing problems $\mathcal{P}^0$ and $\mathcal{P}^1$
7 $\quad p' \leftarrow \text{argmin}_{p \in \mathcal{P}^{\text{RMP}} \cup \mathcal{P}^{heur}}\{\bar{c}_p\}$
8 NON-PROPER-PURE-DIVING$_m(d', p', \mathcal{P}^{part}, \mathcal{P}^*)$

Note also that the completion heuristic has also to be modified. Indeed, when a complete partial solution has to be built, one needs to solve also the subproblem for the residual bin by considering bin $(\tilde{W}, H)$ and hypergraph associated to $\mathcal{P}^0$. This modification is straightforward.

3.2.4 Diving heuristic for the C-2BP$_l$

To solve the C-2BP$_l$ by solving consecutive 2BP$_l$, the subroutine NON-PROPER-PURE-DIVING$_m$ is used as shown in Algorithm 10. In the first step of the algorithm, the number of used bins and the item set to work are initialized (line 1). The main part of the algorithm is iterative. The first step is to obtain the set of available bins. It is enriched with a new bin B' if there is a leftover bin from the previous batch (lines 3-7). The current instance is solved as a 2BP$_l$ using Algorithm 6 (line 8). From the obtained solution for the current item set, the number of used bins is recorded. Note that if the batch is not the first, the number of bins is decreases by one since the leftover bin for this set has already been counted when solving previous batch (lines 9-10). Finally, the algorithm passes on the next batch. The process is repeated until there is no more batch to cut. At the end of the process, a complete solution to the C-2BP$_l$ is obtained as well as the total number of bins to use.

Algorithm 10: Heuristic for the C-2BP$_l$

```
1  nb_bin ← 0, s ← 1
2  do
3      B ← set of available bins
4      if s > 1 then
5          W' ← last bin used length from solution S_{s-1}
6          create a new bin B' = (W − W', H)
7          put B' as the first element in B
8      S_s ← solution of the 2BP_l using B, I_s and
       NON-PROPER-PURE-DIVING_m
9      nb_bin ← nb_bin + |S_s|
10     if s > 1 then  nb_bin ← nb_bin − 1  s ← s + 1
11 while s <= |I|
```

If one is interested in solving the C-2BP, two modifications have to be applied to Algorithm 10. The first one is to remove line 10. The second is to solve the problem at a given iteration as 2BP (line 8). To solve the C-2BP$_l$ by diving heuristic with LDS, the diving heuristic to call at line 8 has to be modified.

3.2.5 Computational experiments

In an industrial context, C-2BP and C-2BP$_l$ are solved iteratively for different item batches. To validate the methodology described in this section, real-world instances with batches are considered. The standard bin dimension to work on is $(W, H) = (6000, 3000)$. Each instance is composed of 10 or 15 batches (*i.e.* $|\mathbf{I}| = 10$ or $|\mathbf{I}| = 15$). Each batch $\mathcal{I} \in \mathbf{I}$ is composed of $|\mathcal{I}_s| \in \{100, 150\}$ items. Average demand of each item is between 2 and 3. An instance class named

$LXIY$ contains instances with X batches, each of which having Y items. Each class is composed of 50 instances.

The retained way to solve both C-2BP and C-2BP$_l$ uses Algorithm 10 from Section 3.2.4. This algorithm iteratively solves the 2BP or 2BP$_l$ related to each item batch in a C-2BP or C-2BP$_l$ instance. The retained way to solve each cutting problem for each batch uses methods described in Section 3.1. From results shown in Section 3.1.4.2, the most efficient are only consider hereinafter:

- the evolutionary algorithm (ea);
- the algorithm (iub), which combines evolutionary algorithm with list heuristics;
- the "non-proper" diving heuristic, denoted (div), with partial enumeration Δ_3 in the pricing problem and with simple evaluation of the residual problem in the diving (parameter $lastPlateOnly = true$);
- a combination of the "non-proper" diving heuristic and the evolutionary algorithm with complete evaluation of the residual problem in the diving (parameter $lastPlateOnly = false$), called ($ediv$).

Results are not reported for heuristic ($div_\varnothing$) as it was shown to be dominated by (div). Results are also not obtained by div_{32} and $ediv_{32}$ since required computation time is very long and obtained solutions are only slightly better than the solutions obtained after only one dive.

In the left part of Table 3.7 and for each heuristic, the average number of bins needed to cut items from all batches are first reported. The average lower bound (lb) is also outlined. This value is obtained by iteratively computing the rounded-up column generation lower bound for each batch and determining the length of the leftover plate for the next bin based on this bound. To measure the impact of solving C-2BP$_l$, in column "w/o lo", the average of the best solution values obtained by the propose methodology for the C-2BP variant is reported. The right part of the table contains the average computation time.

			Solution value				t, min.			
Instances	w/o lo	lb	ea	iub	div	$ediv$	ea	iub	div	$ediv$
L10I100	132.1	124.7	127.8	127.5	125.8	125.7	1	29	24	32
L10I150	174.0	182.3	186.2	186.0	183.6	183.4	3	127	80	119
L15I100	203.0	192.1	196.8	196.5	193.8	193.5	2	46	35	47
L15I150	290.0	281.7	287.6	287.4	283.6	283.4	5	184	127	176

Table 3.7: Comparison of heuristics on the real-world instances with batches

As observed in Table 3.7, solving the C-2BP$_l$ instead of the C-2BP allows one to save up to 10 plates. As shown in Section 3.1.4.2, heuristic (ea) is the

fastest. The more expensive heuristic (*iub*) improves on (*ea*) only marginally. Better results are obtained by diving heuristics. The standard diving heuristic (*div*) saves up to 4 plates or 1.4% of plates on average. The extended diving heuristic (*ediv*) saves up to 4.2 plates or 1.5% of plates on average. Moreover the gap with the lower bound is at most 1.9 plates or 0.8% of plates on average using (*div*). For the (*ediv*) this drops to 1.7 plates or 0.7% on average. The "non-proper" diving heuristic (*div*) offers the best solution quality – running time trade-off. Even if its running time reaches 2 hours for the largest instances, its application in practice is still realistic. Indeed a C-2BP$_l$ instance corresponds to a one day planning horizon. Therefore, spending two hours to obtain a solution seems to be reasonable.

3.3 Conclusion

In this chapter, the 2BP$_l$ was solved with column generation based diving heuristics. An originality of this work is that these heuristics use non-proper cutting patterns. This variant simplifies the pricing problem but makes it more difficult to obtain feasible solutions. Several ways to overcome this difficulty are proposed including combination with an evolutionary algorithm and partial enumeration technique. The latter reduces the number of generated non-proper cutting patterns, tightens the column generation lower bounds and improves the quality of solutions obtained by the diving heuristics. The computational experiments on the literature and real-life instances revealed that the proposed diving heuristics for the 2BP$_l$ outperformed significantly the constructive and evolutionary heuristics. The largest improvements are achieved on real-life large instances. This diving heuristic with non-proper columns is generic and can be applied to other cutting problems. In a second time, the C-2BP$_l$ was studied in order to measure the impact of the diving heuristic when consecutive item batches are cut. The main motivation is to measure savings for a production day. Experiments on real-life production plant instances showed that diving heuristics run in a reasonable time and allow the decision maker to save raw material on average compared to constructive heuristics.

Chapter 4

Bin-packing problems with defects

Works done in previous chapters only consider defect-free cutting problems and does not totally match industrial expectations. From its definition, the structure of the industrial cutting problem, called hereinafter $2BP_{dpl}$, is close to its defect-free relaxation. Thus, one can use solving methods described in Chapter 3. The straightforward way is to reformulate the $2BP_{dpl}$ using Dantzig-Wolfe decomposition and to tackle it with column generation. The main difficulty of such reformulation is that pricing problems to deal with are difficult to solve in a short amount of time. In theory, one can use the generic procedure detailed in Chapter 2 to solve them starting from the unbounded dynamic program for the defect case. Nevertheless, even if it is possible to write such dynamic program, it remains huge in practice as outlined by Afsharian et al. [1]. An attempt was made to write it but the approach failed to have exploiting results. To overcome the pricing problem difficulty, two alternatives are described in this chapter. The first solves the defect-free relaxation of $2BP_{dpl}$. The relaxed solution is modified with post-processing methods to ensure its feasibility regarding $2BP_{dpl}$ constraints. A second way to solve the $2BP_{dpl}$ is to start from the non-proper diving heuristic and to modify it to handle defects in the fixing procedure. Before explanation of solving methods, the $2BP_{dpl}$ is introduced mathematically.

4.1 Problem formulation

From the problem description mentioned in the introduction of this manuscript, the industrial cutting problem is close to $2BP_l$. The extra considerations to keep in mind is that bins may have defects and are stacked in the factory. This allows one to extend the $2BP_l$ notation.

The two-dimensional bin-packing problem with leftover and with precedence constraints between bins that might have defects ($2BP_{dpl}$) is a variant of $2BP_l$ and $2BP_d$. The $2BP_{dpl}$ is an optimization problem which aims to minimize the total length of used bins required to pack a subset of rectangular

items. All bins $b \in \mathcal{B}$ have the same dimension (W, H) but are characterized by a specific set of defects $\mathcal{D}_b$. In practice, defects may not be rectangular but with guillotine cuts, only rectangular pieces will be removed anyway. Each defect $d \in \mathcal{D}_b$ is then defined by two coordinates (x_d, y_d) and a width and height (w_d, h_d). If a defect has a non rectangular form it is approximated by drawing a minimum size rectangle around it. If the defect set of a bin b is empty, the bin is defect free. Each item $i \in \mathcal{I}$ to cut has a fixed width and height (w_i, h_i) and has to be cut exactly d_i times. A solution to such problem is a set of cutting patterns ensuring precedence constraints between bins. Each cutting pattern applied on a bin has to ensure that no items are cut in a defective area and no cuts are made through a defect.

Assume a 2BP_{dpl} instance. Let $\mathcal{P}^1(b)$ be the set of valid patterns without leftover consideration and $\mathcal{P}^2(b)$ be the set of valid patterns with leftover consideration for a bin $b \in \mathcal{B}$. Let binary variable $\lambda_p^b = 1$ if bin b is cut using pattern $p \in \mathcal{P}^1(b) \cup \mathcal{P}^2(b)$, 0 otherwise. Let also binary variable $y_b = 1$ if bin b is used, 0 otherwise. Similarly, binary variable $z_b = 1$ if bin $b \in \mathcal{B}$ is the last bin in the solution, 0 otherwise. Notation a_{ip} is the number of items $i \in \mathcal{I}$ cut in pattern $p \in \mathcal{P}^1(b) \cup \mathcal{P}^2(b), b \in \mathcal{B}$. Let also $n(b)$ be bin $b' \in \mathcal{B}$ which precedes bin $b \in \mathcal{B}$. If bin b is the first bin to use, $n(b)$ is set to $\varnothing$. Note that $p \in \mathcal{P}^2(b)$ has an extra information w_p related to the total width of first stage cuts in the associated pattern p. The ILP formulation of the 2BP_{dpl} is:

$$\min \sum_{b \in \mathcal{B}} \sum_{p \in \mathcal{P}^1(b)} W \lambda_p^b + \sum_{b \in \mathcal{B}} \sum_{p \in \mathcal{P}^2(b)} w_p \lambda_p^b \tag{4.1}$$

$$\text{s.t.} \sum_{b \in \mathcal{B}} \sum_{p \in \mathcal{P}^1(b)} a_{ip} \lambda_p^b + \sum_{b \in \mathcal{B}} \sum_{p \in \mathcal{P}^2(b)} a_{ip} \lambda_p^b = d_i, \quad \forall i \in \mathcal{I} \tag{4.2}$$

$$\sum_{p \in \mathcal{P}^1(b)} \lambda_p^b = y_b, \quad \forall b \in \mathcal{B} \tag{4.3}$$

$$\sum_{p \in \mathcal{P}^2(b)} \lambda_p^b = z_b, \quad \forall b \in \mathcal{B} \tag{4.4}$$

$$\sum_{b \in \mathcal{B}} z_b = 1, \tag{4.5}$$

$$y_b - y_{n(b)} \leqslant 0, \quad \forall b \in \mathcal{B}, n(b) \neq \varnothing \tag{4.6}$$

$$z_b - y_{n(b)} \leqslant 0, \quad \forall b \in \mathcal{B}, n(b) \neq \varnothing \tag{4.7}$$

$$y_b \in \{0,1\}, z_b \in \{0,1\}, \quad \forall b \in \mathcal{B} \tag{4.8}$$

$$\lambda_p^b \in \{0,1\}, \quad \forall p \in \mathcal{P}^1(b) \cup \mathcal{P}^2(b), \quad \forall b \in \mathcal{B} \tag{4.9}$$

Objective function (4.1) sums up the width of all standard bins plus the used length of the last bin. A standard bin is considered to be used completely, its contribution to the objective function is set to W, even if the selected cutting

pattern has a total width smaller than the bin width. Constraints (4.2) ensure to cut all items among selected patterns. Constraint sets (4.3)-(4.4) limit the number of patterns to select for each standard bin and for the last bin to be one. Constraint (4.5) is here to impose that only one bin can be considered as the last one. Constraints (4.6) are classical precedence constraints for standard bins. Constraints (4.7) are precedence constraints between standard bins and the last bin.

Since the previous formulation has an exponential number of variables λ, a classical way to deal with it is to generate them dynamically using column generation. The proposed formulation implies to solve at most $2|\mathcal{B}|$ pricing problems. This comes from the fact that bins are assumed to be distinct from each other in a $2BP_{dpl}$ instance because of their defects. Let π the set of dual variables associated with constraints (4.2) and θ (resp. ι) the set of dual variables associated with the constraints (4.3) (resp. (4.4)). The reduced cost of a variable λ_p^b related to a given bin b is:

$$\zeta(\lambda_p^b) = -\sum_{i \in \mathcal{I}} a_{ip}\pi_i + \begin{cases} W - \theta_b, & p \in \mathcal{P}^1(b) \\ w_p - \iota_b, & p \in \mathcal{P}^2(b) \end{cases} \tag{4.10}$$

The first pricing problem is related to solve a C-2KP-RE-4-r$_d$. The second one is a C-2KP-RE-4-r$_{dl}$ where items have to be packed to the left of the bin.

Note that if lower and upper bounds lb and ub on the number of bins to use are known, variables y_b (resp. z_b) for all bins up to the bin at position lb in the stack can be fixed to one (resp. zero). In the same time the number of maximum bins to consider is set to ub instead of $|\mathcal{B}|$. This contributes to reduce the number of constraints and variables in the model.

The Dantzig-Wolfe decomposition applied to $2BP_{dpl}$ entails solving many pricing problems at each column generation iteration. In theory branch-and-price can be used to find the optimal solution of formulation (4.1)-(4.9). In practice this is difficult to large size of problem instances. Nevertheless since the $2BP_{dpl}$ can be relaxed to the $2BP_l$ but just ignoring defects, one can use solving methods from Chapter 3 to tackle the $2BP_{dpl}$.

4.2 Post-processing methods

The $2BP_{dpl}$ implies to deal with C-2KP-RE-4-r$_d$ and C-2KP-RE-4-r$_{dl}$ as pricing problems. Since it is not possible to solve such problems to optimality in a short computation time for large instances even when defects are not considered, one needs to find another solution approach.

From an observation of the $2BP_{dpl}$ structure, one can see that a straightforward relaxation of this problem is to not considering defects. This relaxation

is equivalent to solve the $2BP_l$, which is solved quickly and efficiently from results of the "non-proper" diving heuristic shown in Chapter 3. Since a problem relaxation is solved instead, the obtained $2BP_l$ solution may be infeasible for the initial $2BP_{dpl}$ instance because defects are not considered. The retained strategy in this section is a repairing procedure of a $2BP_l$ solution to make it feasible for the $2BP_{dpl}$. This section details such procedures.

In the remainder of this section, notation S (resp. $\tilde{S}$) is used to denote a solution to the $2BP_l$ (resp. $2BP_{dpl}$). Notation $\mathcal{P}(S)$ (resp. $\mathcal{P}(\tilde{S})$) denotes the set of patterns in solution S (resp. $\tilde{S}$). Patterns $p \in \mathcal{P}(S)$ (resp. $p \in \mathcal{P}(\tilde{S})$) are assumed to be indexed by q, p_q refers to the q-th pattern in $\mathcal{P}(S)$ (resp. $\mathcal{P}(\tilde{S})$). By convention the pattern of index $q = |\mathcal{P}(S)|$ (resp. $q = |\mathcal{P}(\tilde{S})|$) refers to the pattern related to the last used bin in the associated solution S (resp. $\tilde{S}$). Bins in set $\mathcal{B}$ are also indexed by j. This indexation is used to assign patterns to bin to respect bin precedence constraint in the $2BP_{dpl}$.

4.2.1 Naive reparation

A naive way to transform a solution S into a solution $\tilde{S}$ is to remove items which overlap a defect in S. Indeed after solving the $2BP_l$, the obtained solution S is decomposed into a set of patterns $\mathcal{P}(S)$. Each pattern $p \in \mathcal{P}(S)$ corresponds to a feasible cutting pattern without defect consideration. Since both bins in $\mathcal{B}$ and patterns in $\mathcal{P}(S)$ are indexed, a naive way to obtain an infeasible solution $\tilde{S}$ for the $2BP_{dpl}$ is to assign the q-th pattern from $\mathcal{P}(S)$ to the q-th bin in $\mathcal{B}$. This may produce an infeasible assignment between patterns and bins $\mathcal{P}(\tilde{S})$. Since some items may overlap a defect, a way to obtain a feasible assignment is to simply remove overlapping items and then iteratively cut them starting from the bin of index $|\mathcal{P}(\tilde{S})|$. This is possible since the number of bins to cut all items is assumed to be in a quantity large enough to cut all items.

Let $\hat{\mathcal{I}}$ be a set of items which overlap a defect in patterns stored in $\mathcal{P}(\tilde{S})$ after simple assignment from initial solution $\mathcal{P}(S)$. First all items $i \in \hat{\mathcal{I}}$ from $\mathcal{P}(\tilde{S})$ are removed. This automatically makes all patterns in $\mathcal{P}(\tilde{S})$ feasible regarding defects but not regarding item production constraints. Using the assumption that bins are in a quantity large enough to cut all items, items from $\hat{\mathcal{I}}$ can be cut according to the following procedure. Consider the last pattern $p_{|\mathcal{P}(\tilde{S})|}$ of index $|\mathcal{P}(\tilde{S})|$. This pattern is applied on bin $b_{|\mathcal{P}(\tilde{S})|}$. By using list heuristics as the ones mentioned in Section 1.4.3, one can try to cut items in $\hat{\mathcal{I}}$ from $b_{|\mathcal{P}(\tilde{S})|}$ starting from given pattern $p_{|\mathcal{P}(\tilde{S})|}$. If all items can be cut in $b_{|\mathcal{P}(\tilde{S})|}$ avoiding defects, the pattern set $\mathcal{P}(\tilde{S})$ is feasible for the $2BP_{dpl}$. If there are some remaining items, one needs to create an empty pattern for the bin of index $|\mathcal{P}(\tilde{S})| + 1$ and try to pack the remaining items. To be more generic, this is equivalent to solve a $2BP_{dpl}$ starting from bin of index $|\mathcal{P}(\tilde{S})|$ and using new bins as long as there are items to cut. This repairing heuristic is called PUSHBACK.

To be less myopic than heuristic PUSHBACK, one can exploit holes created in different patterns from $\mathcal{P}(\tilde{S})$. Let still $\hat{\mathcal{I}}$ be the set of items which overlap a defect after simple assignment and all items $i \in \hat{\mathcal{I}}$ are removed from patterns in $\mathcal{P}(\tilde{S})$. Then for each pattern $p_q, q = \{1, \ldots, |\mathcal{P}(\tilde{S})| - 1\}$, using a heuristic, one can try to fill empty spaces with items from $\hat{\mathcal{I}}$. Finally the solution is made feasible using heuristic PUSHBACK. This extended repairing process is called FILLING.

Heuristics PUSHBACK and FILLING rely on the fact that a subroutine can find a packing for a given pattern p using only items in $\hat{\mathcal{I}}$. This subroutine uses modified version of Best-Fit, Bottom-Left-Fit, First-Fit, Next-Fit list heuristics for different order of items in $\hat{\mathcal{I}}$. For each item, its profit is equal to its area. In the standard heuristics, *i.e.* without defect consideration, a chosen item is always packed in the bottom left position of a given plate. The built pattern also ensures to be at most four stages from cutting constraints. In the modified version, this check is enhanced to deal with defects. Instead of only checking if an item can be packed in the bottom left corner of the plate, bottom right, top left and top right corners are also considered. This gives more flexibility to the heuristic and increases the odds to pack an item. The idea is to apply list heuristics on $\hat{\mathcal{I}}$ when items are ordered differently. At each iteration, the item set $\hat{\mathcal{I}}$ is randomly sorted. Then for the given order on $\hat{\mathcal{I}}$, Best-Fit, Bottom-Left-Fit, First-Fit and Next-Fit heuristics are called. The obtained result is a pattern p' and a set of packed items $\hat{\mathcal{I}}' \subseteq \hat{\mathcal{I}}$. During the run of list heuristics, the best pattern and associated packed items are stored.

4.2.2 Reparation by subplates permutation

A way to make an infeasible cutting pattern, *i.e.* when items overlap defects, feasible is to use the structure of this cutting pattern. Indeed since cuts are staged, a given cutting pattern can be decomposed into a set of vertical strips obtained after first stage cuts. Each of these strips can itself be divided into horizontal strips after second stage cuts and so on ... This gives a tree representation of a cutting pattern. Each node in this tree is a plate with bottom left corner at point (x, y) with dimension $w \times h$ obtained after a cut of stage s. The root of this tree is the initial plate $(0, 0, W, H, 0)$. An example of the tree representation is given in Figure 4.1.

Remember that $\mathcal{P}(\tilde{S})$ is obtained by assigning the q-th pattern from $\mathcal{P}(S)$ to the q-th bin in $\mathcal{B}$. The pattern set $\mathcal{P}(\tilde{S})$ may contain infeasible patterns. However, a feasible pattern contains subplates where no overlaps occurs. Using the tree representation, one can identify them as depicted in Figure 4.1. Consequently for a given pattern, some subplates are valid because they are cut at the right coordinates in the bin. From the tree representation and using the fact that cuts are staged, it is possible to turn an infeasible pattern into a feasible one by swapping some subplates inside it. This corresponds to

exchange coordinates of some subtrees.

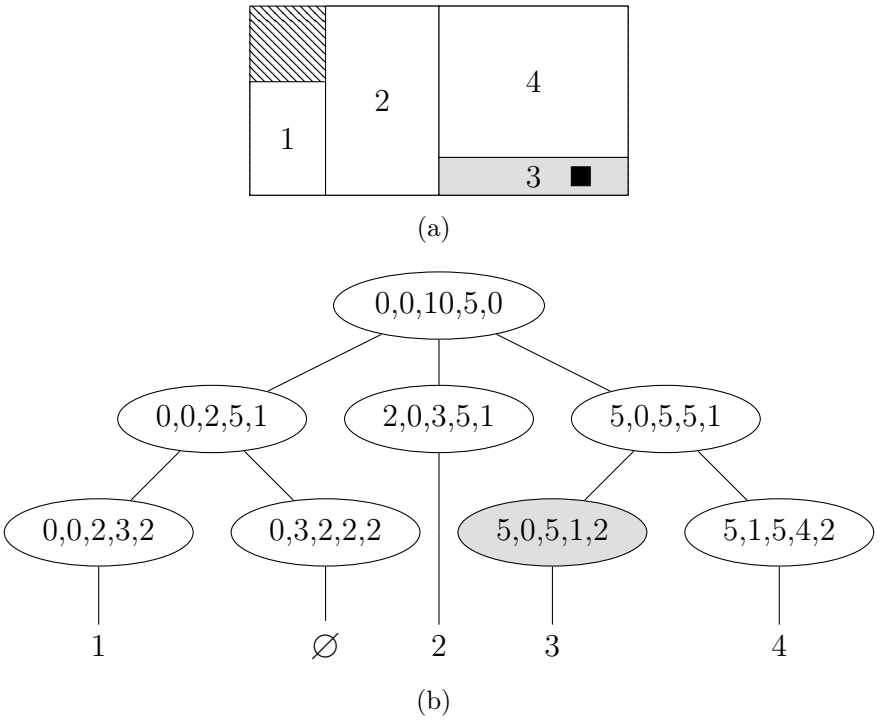

Figure 4.1: Example of a cutting pattern with a defect (a) and its tree representation (b). Each node represents a plate (x, y, w, h, s). The pattern is infeasible since item 3 overlaps a defect.

Let $T(p)$ be the tree representation of pattern $p \in \mathcal{P}(\tilde{S})$. Each node $v \in T(p)$ has a father $h(v)$ and one or more children $\mathcal{T}(v)$. A node v is related to a plate of coordinates $(x_v, y_v) = (x, y)$ with dimension $w_v \times h_v = w \times h$ obtained after a cut of stage $s_v = s$. A node v has also a value p_v computed as the sum of the profits of cut items in the subtree of v which do not overlap a defect. The root node of $T(p)$ is denoted by r and by definition $h(r) = \varnothing$. The permutation procedure recursively swaps children $\mathcal{T}(v)$ of a given node $v \in T(p)$ to ensure feasibility regarding defects. Each node in $\mathcal{T}(v)$ is considered to be distinct. A permutation ς of a given son set $\mathcal{T}(v)$ is a sequence of all nodes in $\mathcal{T}(v)$. Each permutation ς of $\mathcal{T}(v)$ corresponds to a cutting pattern in which coordinates are updated accordingly. By convention, coordinates from nodes $\{v_1, v_2, \ldots, v_{|\mathcal{T}(v)|}\} \in \varsigma$ are updated from left to right if v is obtained by a cut of even stage or v is the root node r. In other words, the leftmost node v_1 in ς has coordinates (x_v, y_v), the second one v_2 has coordinates $(x_v + w_{v_1}, y_v)$, the third one v_3 has coordinates $(x_v + w_{v_1} + w_{v_2}, y_v)$ and so on. Note that coordinates of descendants of each node in ς have also to be updated. Nodes in ς are updated from down to top if v is obtained by a cut of odd stage. The leftmost node v_1 has coordinates (x_v, y_v), the second one v_2 has coordinates

$(x_v, y_v + h_{v_1})$, the third one v_3 has coordinates $(x_v, y_v + h_{v_1} + h_{v_2})$ and so on.

A way to compute the set of all permutations of $\mathcal{T}(v)$, denoted by $\sigma(\mathcal{T}(v))$, is to use a brute force enumeration. Nevertheless some permutations can be considered as equivalent under certain assumptions. Consider a node $v \in T(p)$ and a permutation $\varsigma \in \sigma(\mathcal{T}(v))$ with order on nodes $\{v_1, v_2, \ldots, v_k, \ldots, v_{|\mathcal{T}(v)|}\}$. Consider that s_v is even. Using permutation ς, the x-coordinate of fixed node v_k is $x_{v_k} = x_v + \sum_{l=1}^{k-1} w_{v_l}$. Now assume that the bin associated to the current pattern p has no defects inside the rectangle between coordinates $[x_v, x_{v_k}]$ and $[y_v, y_v + h_v]$. Consequently the order of nodes $\{v_1, v_2, \ldots, v_{k-1}\}$ does not have any impact on defects overlapping. From this observation, all permutations $\varsigma' \in \sigma(\mathcal{T}(v)), \varsigma' \neq \varsigma$ such that the order on nodes of ς' up to the $k-1$-th node is a permutation of $\{v_1, v_2, \ldots, v_{k-1}\}$ in ς can be discarded. Equivalently, this observation also holds when s_v is odd and the bin has no defects inside the rectangle between coordinates $[x_v, x_v + w_v]$ and $[y_v, y_{v_k}]$ with $y_{v_k} = y_v + \sum_{l=1}^{k-1} h_{v_l}$ for $\varsigma \in \sigma(\mathcal{T}(v))$.

Algorithm 11: PERMUTATION(v)

```
(val_best, ς_best) ← (0, ∅)
compute set σ(T(v)) of permutations
for ς ∈ σ(T(v)) do
    for v' ∈ ς do
        if there is an overlapping in v' then PERMUTATION(v')
    val ← Σ_{v'∈ς} val_{v'}
    if val > val_best then (val_best, ς_best) ← (val, ς)
update subtree of v using ς_best
```

From the definition of permutation between strips, a recursive procedure, called PERMUTATION(v) and outlined in Algorithm 11, is used to find the permutation of best value for a given pattern p. The permutation procedure uses as input a node $v \in T(p)$. First, the best permutation and its value $(val_{best}, \varsigma_{best})$ are set to zero. The set of all permutations $\sigma(\mathcal{T}(v))$ is then computed with a brute force enumeration. Equivalent patterns are used to reduce the number of permutations by removing equivalent ones. For each permutation $\varsigma \in \sigma(\mathcal{T}(v))$, one first checks if the order of nodes in ς does not lead to an overlap between items and defects. If an overlap is detected for a node $v' \in \varsigma$, PERMUTATION(v') is called. Then for the current permutation ς, its value val is computed by summing up the value of each node $v' \in \varsigma$. During exploration of all $\varsigma \in \sigma(\mathcal{T}(v))$, the permutation of best value is stored. Finally the node v is updated according to node order in ς_{best}. By definition, since the amount of wasted area in a cutting pattern has to be minimized, the value of nodes v in the tree representation $T(p)$ is linked to the area of cut items. For all terminal nodes v in the tree representation $T(p)$, if the

node leads to a waste or an item cut but overlaps a defect, its value is zero, its value is the area of the cut item otherwise. Pattern from Figure 4.1 after permutation is depicted in Figure 4.2.

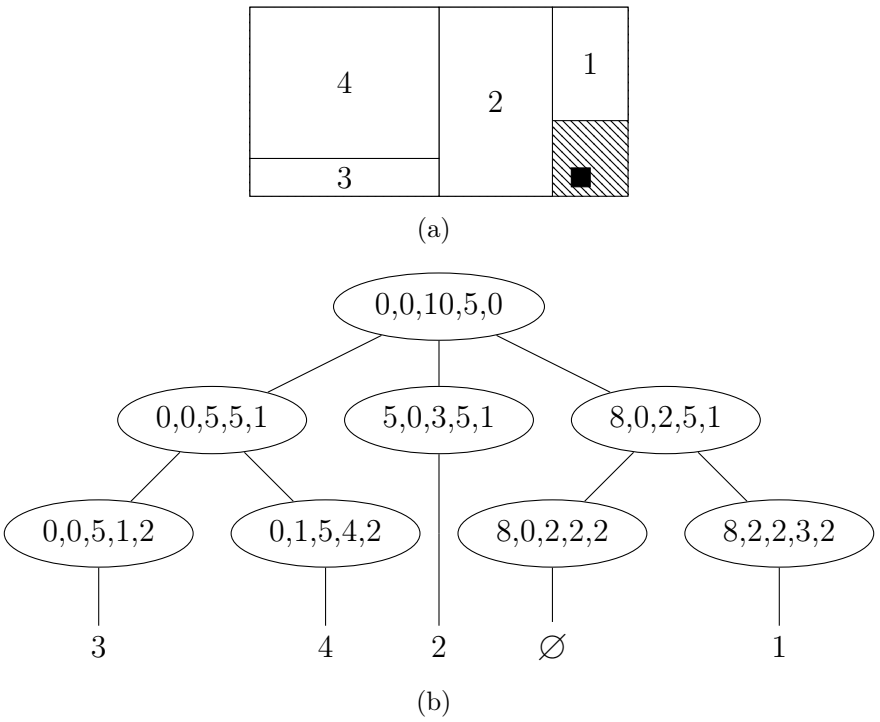

Figure 4.2: Transformation of the infeasible pattern from Figure 4.1 into a feasible one with strip permutation.

From the permutation which can be performed inside a pattern p, one can define a repairing heuristic to make patterns in $\mathcal{P}(\tilde{S})$ feasible. For each pattern $p \in \mathcal{P}(\tilde{S})$, one has to represent it as a tree $T(p)$ rooted at r and then calls PERMUTATION(r). The permutation subroutine has two properties. The first one is that if there is a permutation which turns the pattern into a feasible one, the subroutine will find it. The second property is that if no feasible permutation exists, the pattern will be rearranged such that the set of items which overlap defects has the minimum total area. If the pattern remains infeasible after permutation, the procedure FILLING is used to make it feasible.

Using the permutation subroutine, one can design a more global method. For all previous repairing heuristics, it is supposed that patterns in $\mathcal{P}(\tilde{S})$ are already assigned to a bin. One can avoid to be dependant of this prefixed order by evaluating, for each pattern $p \in \mathcal{P}(S)$ and each bin $b \in \mathcal{B}$, the value of the best permutation when pattern p is applied on bin b. This can be represented as an assignment problem between patterns and bins.

Let $G = (\mathcal{V}_p, \mathcal{V}_b, \mathcal{A})$ be a bipartite graph. $\mathcal{V}_p$ contains nodes p_q referring

to the pattern of index $q = \{1, \ldots, |\mathcal{P}(S)|\}$. $\mathcal{V}_b$ contains nodes b_j referring to the j-th bin in $\mathcal{B}$, $j = \{1, \ldots, |\mathcal{P}(S)|\}$. The edge set $\mathcal{A}$ is composed of edges connecting a node in $\mathcal{V}_p$ to a node in $\mathcal{V}_b$. Each edge $a \in \mathcal{A}$ has a weight w_a equal to the value of the best permutation when bin b_j is cut using pattern p_q. A feasible assignment between patterns and bins is obtained by finding a matching of size $|\mathcal{P}(S)|$ in graph G. The maximum weighted matching corresponds to the best assignment. Such problem can be solved in polynomial time using algorithms from Edmonds [26].

The edge set in the obtained maximum weighted matching is then used to assign each pattern p_q to each bin b_j. This gives a new way to create the pattern set $\mathcal{P}(\tilde{S})$ and a solution $\tilde{S}$. Note that some patterns in $\mathcal{P}(\tilde{S})$ can still be infeasible. Consequently, a reparation heuristic has to be used. The procedure to define the matching problem is given in Algorithm 12.

Algorithm 12: Matching heuristic

```
G = (V_p, V_b, A) = (∅, ∅, ∅)
for j = 1 to |P(S)| do
    V_p ← V_p ∪ {p_j}, V_b ← V_b ∪ {b_j}
    for j' = 1 to |P(S)| do
        a ← edge between p_j and b_j'
        A ← A ∪ {a}
        p' ← apply pattern p_j on bin b_j'
        r ← root of tree representation of p'
        PERMUTATION(r)
        w_a ← value of best permutation of p'
find maximum weighted matching on G
P(S̃) ← assign patterns to bins using the matching
FILLING(P(S̃), ∅)
```

4.2.3 Reparation by solving the 1BP

When one wants to assign patterns from $\mathcal{P}(S)$ to bins with defects, there is another way to exploit the recursive structure of the patterns. A given pattern $p \in \mathcal{P}(S)$ can be decomposed into a set of vertical strips $\tau(p)$ of different widths and fixed height H. Let τ be the set of all vertical strips obtained after first stage cuts among all patterns $p \in \mathcal{P}(S)$, $\tau = \bigcup_{p \in \mathcal{P}(S)} \tau(p)$. Using strips in τ, the $2BP_{dpl}$ can be transformed as a $1BP_{dpl}$ in which items to assign are strips from τ. This restriction aims to assign strips in τ to different bins with defects. As in the classical 1BP, each strip can be assigned to only one bin and total width of strips assigned to a bin cannot exceed the bin width. An interest of this restriction is to break the structure of initial patterns from $\mathcal{P}(S)$ and then recombine the different vertical strips together in order to create new cutting

patterns.

For example, one can use a fast heuristic to obtain an assignment of strips to bins. If this assignment is valid (non-overlapping defects), one has obtained a feasible solution to the $1BP_{dpl}$ and consequently to the $2BP_{dpl}$. For this, a modified First-Fit heuristic is used. It takes a given order on strips τ and assigns them to bins without taking into account defects. This assignment is then evaluated: total value of items overlapping defects is computed. To diversify obtained assignments, the First-Fit heuristic is called with different orders on τ as input. This building phase is repeated until a threshold on the number of different orders on τ is reached. The best assignment is stored until the threshold is reached. Note that this best assignment may still be infeasible. To make the solution feasible, the function FILLING is then called. The procedure is described in Algorithm 13 and one has to called FIRST-FIT($\mathcal{P}(S)$) to start it.

Algorithm 13: FIRST-FIT($\mathcal{P}(S)$)

$\tau \leftarrow \bigcup_{p \in \mathcal{P}(S)} \tau(p)$
$(val_{best}, \mathcal{P}(\tilde{S})) \leftarrow (0, \varnothing)$
for $it = 0$ *to* $maxIt$ **do**
    randomly sort τ
    $(val, \mathcal{P}) \leftarrow$ FirstFit(τ)
    if $val > val_{best}$ **then** $(val_{best}, \mathcal{P}(\tilde{S})) \leftarrow (val, \mathcal{P})$
FILLING($\mathcal{P}(\tilde{S}), \varnothing$)

4.3 Diving heuristic with non-proper columns

Post-processing methods from Section 4.2 are myopic since they are highly dependent on the solution found for the $2BP_l$ in order to find a feasible solution for the $2BP_{dpl}$. An alternative can be to start from the diving heuristic with "non-proper" columns used to solve the $2BP_l$. Computational experiments in Section 3.1.4 outline good solution quality when "non-proper" columns are considered. At the same time, the heuristic runs in a short computation time.

However, to be used to solve a $2BP_{dpl}$ instance, here are two main particularities to have in mind. The first one is that the set of valid cutting patterns $\mathcal{P}(b)$ for a given bin $b \in \mathcal{B}$ is in general different from $\mathcal{P}(b'), b' \in \mathcal{B}, b \neq b'$ since bins may have different defects. A cutting pattern can be valid for a bin but not valid for another one. The second particularity is that bins are ordered using a stack. If a pattern is used for a bin $b \in \mathcal{B}$, it implies that valid patterns should exist for all bins stacked above bin b.

In theory, one can directly use the diving heuristic used for the $2BP_l$ to solve the $2BP_{dpl}$. This implies to write a dynamic program for each pricing problems, *i.e.* each bin. In practice, this is impossible due to the huge size of

each dynamic program as outlined by Afsharian et al. [1]. An attempt was made to write them but the approach failed to give exploiting results.

The retained approach is start from the diving heuristic for the defect-free problem and slightly modify it. However, the structure of the pricing problem should be kept as simple as possible. The main discussion here is on how to modify the diving heuristic to handle $2BP_{dpl}$ constraints and how to find feasible columns to fix. The diving heuristic for the problem with defects is first described. Second, the way to fix columns is outlined. This section ends with a description of the completion heuristic when bins with defects are considered.

4.3.1 "Non-proper" diving heuristic

To be use for the $2BP_{dpl}$, the fixing subroutine of the "non-proper" diving heuristic, described in Section 3.1.2.1, is modified. In the diving heuristic for the $2BP_l$, bins are defect-free and thus a cutting pattern is valid for all of them. In a $2BP_{dpl}$ instance, bins are assumed to be distinct due to their defect sets. The modification to perform on the diving heuristic is related to the way to fix columns. An extra decision to take is to find a column and assigns it to a bin $b \in \mathcal{B}$. To let pricing problem structure as simple as possible, it is solves without defect consideration.

Let j be the index of bins in $\mathcal{B}$, b_j refers to the j-th bin in $\mathcal{B}$. Let variable b^p be equal to j if pattern p is assigned to bin b_j. By convention for a given pattern p, the value of b^p is 0 if the pattern is not assigned to a bin. The column fixing procedure uses as input a set of patterns such that for each pattern p, $b^p = 0$. The process tries to find an assignment of a pattern p to an available bin. If such assignment is found, variable b^p is fixed. As the number of fixed patterns increases, the number of available bins becomes smaller. To reduce the search of an assignment of a given pattern p to an available bin, a lower bound lb should be first computed to know the minimum number of bins in a valid solution for the $2BP_{dpl}$. Note that if the lower bound quality is not good, the diving can be stuck in that all bins up to the bin of index lb are fixed but the fixing procedure cannot provide a new assignment. To avoid this, the fixing procedure can fix sequentially the bin of index $lb + 1$, $lb + 2$ and so on. This two step fixing is mandatory because of the completion heuristic. Indeed, the completion heuristic only considers bins of indices $lb + 1$ and greater when trying to build a feasible solution. The idea is to do the assignment of mandatory bins in the fixing procedure and assignment of extra bins is done by the completion heuristic.

The proposed modified "non-proper" diving heuristic for the problem with bins with defects is represented in Algorithm 14. Let assume that $\mathcal{J}$ is the set of available bin indices. Since a primal feasible solution $\mathcal{P}^*$ is known, $\mathcal{J} = \{1, \ldots, |\mathcal{P}^*|\}$. Let set $\tilde{\mathcal{J}}$ containing bin indices $\{1, \ldots, lb\}$. The first step of the heuristic is to update the partial solution with the previous fixed pattern

(line 2). The second step of the algorithm is the completion heuristic (lines 3-5). Following the completion heuristic, the residual master problem is solved (line 6). Set $\mathcal{P}$ of feasible patterns is then obtained from the residual master problem and given to the fixing subroutine. The fixing subroutine uses $\mathcal{P}$ and also the set of available bins. It returns the pattern p' which has to be fixed (line 7). Note that if $\mathcal{P}$ is empty, the fixing procedure generates a heuristic pattern. This is mandatory to ensure that a valid cutting pattern is fixed. At the end of the fixing process, the method is called recursively (line 8).

Algorithm 14: NON-PROPER-PURE-DIVING$_d(d, p, \mathcal{P}^{part}, \mathcal{P}^*, \mathcal{J}, \tilde{\mathcal{J}})$

1 **if** $p \neq \varnothing$ **then**
2 $\quad d' \leftarrow d - a^p$, $\mathcal{J}' \leftarrow \mathcal{J} \backslash \{b^p\}$, $\tilde{\mathcal{J}}' \leftarrow \tilde{\mathcal{J}} \backslash \{b^p\}$, $\mathcal{P}^{part} \leftarrow \mathcal{P}^{part} \cup \{p\}$
3 $\mathcal{P}^r \leftarrow$ COMPLETION-HEURISTIC$_d(d', lastPlateOnly, \mathcal{J}')$
4 **if** $d - \sum_{p' \in \mathcal{P}^r} a^{p'} = \mathbf{0}$ *and cost* $(\mathcal{P}^{part} \cup \mathcal{P}^r) < cost(\mathcal{P}^*)$ **then**
5 $\quad \mathcal{P}^* \leftarrow \mathcal{P}^r \cup \mathcal{P}^r$
6 solve the RMP of (3.1)-(3.5) with demand bounds d' by column generation and store the set of proper patterns $\mathcal{P}$
7 $p' \leftarrow$ FIXING$(d', \mathcal{P}, \mathcal{J}, \tilde{\mathcal{J}}, \varnothing)$
8 NON-PROPER-PURE-DIVING$_d(d', p', \mathcal{P}^{part}, \mathcal{P}^*, \mathcal{J}', \tilde{\mathcal{J}}')$

Note that the last parameter of procedure FIXING is empty. This is related to the fact that this parameter is used for the tabu list management for the diving heuristic with LDS. The latter is not outlined here.

4.3.2 Column fixing

The key ingredient of the "non-proper" diving heuristic is the column fixing. Indeed it has a major influence on the search in the diving tree. It is performed under assumption that a set of cutting patterns $\mathcal{P}$, a set $\mathcal{J}$ of available bin indexes $\{1, \ldots, ub\}$ and a set $\tilde{\mathcal{J}}$ of bin indexes $\{1, \ldots, lb\}$ are given as input of the fixing subroutine. ub (resp. lb) is an upper (resp. a lower) bound on the number of bins to fix. $\mathcal{P}$ may contain non-proper patterns.

The proposed fixing procedure is the following. For each available bin $j \in \tilde{\mathcal{J}}$, a set of heuristic patterns is obtained by the evolutionary algorithm. Then each proper pattern $p \in \mathcal{P}$ is assigned to bin b_j and rearranged by the permutation subroutine. Remark that function PERMUTATION does not necessarily output a feasible pattern. To make the pattern feasible, all items which overlap defects are removed from it. The pattern obtained after a permutation is then stored (lines 1-9). Secondly, if there are not enough created patterns compared to the tabu list size, heuristic ones are created for the first available bin in $\mathcal{J}$ (lines 9-12). The choice to consider only the first available bin is to be able to create enough different patterns for that bin to help the diving going ahead. After that, the set of created patterns is then

sorted and the best one is returned.

Algorithm 15: FIXING($d, \mathcal{P}, \mathcal{J}, \tilde{\mathcal{J}}$)

```
1  P^fix ← ∅
2  for j ∈ J̃ do
3      P^fix ← P^fix ∪ patterns found by the evolutionary algorithm applied
       for bin b_j with defects and with demand d
4      for p ∈ P do
5          p' ← copy of pattern p assigned to bin b_j
6          r ← root of tree representation of p'
7          PERMUTATION(r)
8          remove overlapping items from p'
9          P^fix ← P^fix ∪ {p'}
10 if P^fix = ∅ or not enough patterns regarding Z then
11     j' ← min_{j∈J}{j}
12     P^fix ← Pset^fix ∪ patterns from evolutionary algorithm for bin b_{j'}
       with defects and with demand d
13 P^fix ← P^fix \ Z
14 SORT(P^fix)
15 p_best ← pattern of best value in P^fix
16 return p_best
```

There are several ways to select the pattern to fix in SORT function. Let p be a pattern from the available set of patterns to fix $\mathcal{P}^{fix}$. Function $val(p)$ gives the value of pattern p computed depending on the set of items it contains. As mentioned in the basic fixing procedure, the value of an item is either its reduced cost or its area. This changes the value of a pattern and possibly the fixed bins during diving heuristic.

The intuitive way to find the pattern to fix is to sort them by non-decreasing values . If two patterns have the same value, the selected pattern is the one associated to the bin with the maximum number of avoided defects. This selection process is also naive since patterns assigned to defect-free bins will probably be chosen first.

A simple statement regarding patterns is that a given cutting pattern is always valid for a defect-free bin but it usually requires some rearrangements to make it feasible for a bin with defects. Thus, the difficulty of finding a cutting pattern of a good value for a bin with many defects is harder. This observation suggests a third way to select a pattern. Here both its value and the number of avoided defects are used to sort them. Patterns are sorted in non-decreasing order by their values times the number of avoided defects.

The previous sorting procedures consider both defect-free bins and bins with defects at the same time. It is more "difficult" to find a good value pattern for a bin with defects. One can consider them first and then the

defect-free ones since all patterns are valid for defect-free bins. The retained way to do it is a two-step sorting of patterns. First all patterns attached to a bin with defects are sorted in non-decreasing order of their values. The one of best value is then fixed. Second if there are no bins with defects, all patterns attached to a defect-free bin are sorted in non-decreasing order of their values and the one of best value if then fixed.

4.3.3 Completion heuristic

The aim of the completion heuristic used in Algorithm 14 is to build a good quality cutting pattern for a set of bins. It is done here in two phases. The completion heuristic is formalized in Algorithm 16.

The first phase is based on the completion heuristic for the defect-free case (lines 6-12). For a given bin b, a set of patterns is built using the hypergraph heuristic and the evolutionary algorithm. If the bin has at least one defect, the permutation subroutine is called on each pattern to try to make it feasible if needed. At the end of the first step, patterns may be infeasible.

The second phase is to directly take into account defects for a given bin b (line 13). This is achieved by calling the evolutionary algorithm adapted for the case with defects. When a subplate (w, h) is defect-free, an item $i \in \mathcal{I}$ such that $w_i \leqslant w, h_i \leqslant h$ can always be cut in its bottom left corner. When a subplate contains defects, items are cut also from the top left corner, bottom right corner or top right corner.

After the second phase, the set of infeasible patterns is discarded. Then the pattern of best value is kept and is used for bin b (lines 15-18). The set of obtained patterns $\mathcal{P}^r$ is then returned as a valid partial solution.

If one is interested in finding a solution to the pricing problem $\mathcal{P}^2$, *i.e.* when *lastPlateOnly* is set to true in Algorithm 14, only lines 6-15 from Algorithm 16 have to be called. The hypergraph to consider is the one related to $\mathcal{P}^2$ instead.

Algorithm 16: COMPLETION-HEURISTIC$_d(d, lastPlateOnly, \mathcal{J})$

$\mathcal{P}^r \leftarrow \varnothing$, $A \leftarrow \sum_{i\in\mathcal{I}} w_i h_i d_i$
$G \leftarrow$ hypergraph related to $\mathcal{P}^1$
do
  $\mathcal{P}' \leftarrow \varnothing$
  if $A \leqslant W \times H$ **then** $G \leftarrow$ hypergraph related to $\mathcal{P}^2$
  $j \leftarrow \min_{j'\in\mathcal{J}}\{j'\}$
  $p_1 \leftarrow$ pattern from evolutionary algorithm on bin b_j with demand d
  $p_2 \leftarrow$ pattern using hypergraph heuristic
  $\mathcal{P}' \leftarrow \mathcal{P}' \cup \{p_1\} \cup \{p_2\}$
  for $p' \in \mathcal{P}'$ **do**
    $r \leftarrow$ root of tree representation of p'
    PERMUTATION(r)
    if p' *infeasible for bin* b_j **then** $\mathcal{P}' \leftarrow \mathcal{P}'\backslash\{p'\}$
  $p_3 \leftarrow$ pattern from evolutionary algorithm on bin b_j with defects and with demand d
  $\mathcal{P}' \leftarrow \mathcal{P}' \cup \{p_3\}$
  $p \leftarrow \text{argmax}_{p'\in\mathcal{P}'}\{val(p')\}$
  $\mathcal{J} \leftarrow \mathcal{J}\backslash\{b^p\}$
  $d \leftarrow d - a^p$, $A \leftarrow A - \sum_{i\in\mathcal{I}} w_i h_i a_i^p$
  $\mathcal{P}^r \leftarrow \mathcal{P}^r \cup \{p\}$
  if $lastPlateOnly$ **then** break
while $d \neq 0$
mark all patterns in $\mathcal{P}^r$ as related to $\mathcal{P}^1$
mark the last inserted pattern in $\mathcal{P}^r$ as related to $\mathcal{P}^2$
return $\mathcal{P}^r$

4.4 Computational experiments

This section reports results from computational experiments to solve the 2BP$_{dpl}$. Used datasets used are taken from industrial data. The bin size in an instance is set to $(6000, 3000)$, the number of different items $|\mathcal{I}|$ can be either 100 or 150. The average demand of an item is two. Since the number of defects is not known a priori, it is difficult to have an estimation of their positions and sizes. To simulate such behaviour and test the limit of described approaches, defects are generated randomly. The number of defects for each bin is obtained using a Poisson distribution with parameter μ. Different values of μ are used to create more or less defects $\{0.33, 1, 3\}$. The size of defects is randomly generated in interval $[1, 10]$ for both dimensions. A dataset named $I100D0.33$ corresponds to an instance with $B = (6000, 3000)$, $|\mathcal{I}| = 100$ and the Poisson distribution set with parameter $\mu = 0.33$. Respectively, a dataset named $I150D3$ is an

instance with $B = (6000, 3000)$, $|\mathcal{I}| = 150$ and $\mu = 3$. Each created dataset contains 50 instances, this gives a total of 150 instances.

The goal of lead experiments for the $2BP_{dpl}$ is twofold: (i) to evaluate the quality of post-processing methods (ii) to evaluate the impact of the different variants of column fixing

All experiments are run using a 2.5 Ghz Haswell Intel Xeon E5-2680 with 128Go of RAM. CPLEX 12.6 is used to solve linear programs. The time limit to solve one instance is set to two hours.

4.4.1 Impact of post-processing methods

As mentioned previously, a way to solve the $2BP_{dpl}$ is to initially solve its relaxation by not taking defects into account. Such relaxation is equivalent to solve the $2BP_l$. In the following part of this section, the retained method to solve the $2BP_l$ relaxation is the "non-proper" diving heuristic (div). This choice is motivated by its good tradeoff between solution quality and computation time as outlined in Section 3.1.4.2. To obtain a feasible solution for the $2BP_{dpl}$, all post processing methods outlined in Section 4.2 are used:

- PUSHBACK heuristic (pb);
- FILLING heuristic (fi);
- PERMUTATION subroutine followed by FILLING heuristic (pe);
- matching representation outlined in Algorithm 12 (ma);
- FIRST-FIT heuristic (ff)

The evolutionary algorithm modified to handle bin with defects is also tested (ead). Results are reported in Table 4.1. In the right part of the table, the reported value is the average gap from the value of the column generation for the $2BP_l$. The left part outlines the average computation time for each method.

	gap,%						t,sec.					
Instances	ead	pb	fi	pe	ma	ff	ead	pb	fi	pe	ma	ff
I100D0.33	2.8	4.2	3.6	2.1	1.5	2.1	7	118	116	117	118	120
I100D1	3.1	9.5	6.4	3.8	2.8	3.8	7	129	130	130	131	148
I100D3	4.4	22.3	14.7	8.3	6.0	8.2	8	129	129	130	161	366
I150D0.33	2.4	3.6	3.0	1.8	1.4	1.9	21	443	446	437	441	457
I150D1	2.6	9.5	6.1	3.1	2.0	3.1	22	431	440	432	439	495
I150D3	3.8	23.8	14.0	7.2	5.3	7.3	24	461	459	461	478	732

Table 4.1: Comparison of post processing heuristics

In Table 4.1, naive heuristics (pb) and (ffi) have the worst solution quality. Using permutation subroutine with filling heuristic (pe) provides better solutions but the gap remains high in particular for instances with many defects per bin. Best results are obtained using the matching representation (ma). Computation time for all post processing methods have the same order of magnitude except for heuristic (ff). This heuristic has the longest computation time due to the threshold value (set to $|\mathcal{I}|$). Post-processing heuristic (ma) has a good trade off between solution quality and computation time. Nevertheless when compared to (ead), solution quality is good for datasets D0.33 and D1. When there are more defects on bins, the evolutionary algorithm handling bins with defects is preferred. One can also observe that heuristic (ead) has always the smallest computation time.

It can be observed from those results that for instances with many defects per bin, post processing methods are not the best approach to use. However, for instance with a small number of defects per bin, post processing with matching representation is preferred.

4.4.2 Impact of modified diving heuristic

Instead of using post-processing heuristics, the diving heuristic can be modified to handle bin with defects by modifying the column fixing procedure. Table 4.2 outlines results using column fixing procedure from Section 4.3. Retained ways to fix columns are:

- fix the one of best value (f^1)
- fix the one of best value times number of avoided defects (f^2)
- first fix the one of best value among bins with defects. If all bins are defect-free, fix the one of best value (f^3)

Notations (f_a) and (f_π) denote the way to fix columns using item areas or reduced costs. Results are also reported for the constructive heuristic (ead), its variant with a longest computation time (lh) and the best post-processing heuristic (ma). The table is divided in two parts; the left one shows average gap from the value of the column generation for the $2BP_l$; the right one the average computation time in seconds.

Results in Table 4.2 outline that doing more iterations of the constructive heuristic (lh) improves slightly results compared to the standard constructive heuristic (ead). The column fixing procedure (f) outperforms the results obtained by constructive heuristics no matter the way to select columns. When the value of each item is equal to its area, the best method is to select columns as done with procedure (f_a^2). Its also outperforms heuristic (ma) for all datasets except I100D0.33 where results are close. Computation time

	gap,%									t,sec.								
Instances	ead	lh	ma	f_a^1	f_a^2	f_a^3	f_π^1	f_π^2	f_π^3	ead	lh	ma	f_a^1	f_a^2	f_a^3	f_π^1	f_π^2	f_π^3
I100D0.33	2.7	2.7	1.5	1.7	1.7	1.7	1.8	1.7	1.8	6	128	118	149	139	140	143	140	139
I100D1	3.1	3.0	2.8	2.1	2.2	2.4	2.3	2.1	2.4	6	128	131	203	175	182	188	176	184
I100D3	4.3	4.3	6.0	3.9	3.8	3.8	3.8	3.8	3.8	7	167	161	303	249	293	333	254	324
I150D0.33	2.4	2.3	1.4	1.4	1.4	1.5	1.6	1.3	1.6	21	587	441	583	516	508	533	526	525
I150D1	2.6	2.5	2.0	1.7	1.5	1.8	1.9	1.4	1.9	21	666	439	817	648	689	729	656	698
I150D3	3.8	3.9	5.3	3.5	3.5	3.4	3.5	3.4	3.5	24	815	478	1225	924	1097	1181	981	1156

Table 4.2: Comparison of fixing procedure

is reasonable since it takes around 15 minutes to find a solution for hardest dataset I150D3. When the value of each item is equal to its reduced cost, the best method is to select columns as done with procedure (f_π^2). As previously, it outperforms heuristic (ma) and has a reasonable computation time.

Results and computation times obtained by methods (f_a^2) and (f_π^2) are close to each other. Consequently there is no real "best" method to select. Nevertheless, the column fixing procedure handling bins with defects ensures good quality results and has to be preferred to constructive or post-processing heuristics.

4.5 Conclusion

In this chapter, two ways to solve the the $2BP_{dpl}$ have been introduced. The first one used different post-processing heuristics. The $2BP_{dpl}$ is relaxed by removing defect consideration and this leads to solve a $2BP_l$ instead. The $2BP_l$ is solved with the "non-proper" diving heuristic detailed in Chapter 3. Since the obtained solution may be infeasible for the initial $2BP_{dpl}$, post-processing heuristics are used. They match patterns and bins with defects by modifying pattern structure. The second way to solve the $2BP_{dpl}$ is to use the "non-proper" diving heuristic with a modified column fixing step. The idea is to use cutting patterns obtained during column fixing and assign them to bins with defects on the fly. Both methods are computationally compared with each other on a set of real industrial instances. They are competitive to solve the $2BP_{dpl}$ but best results are obtained with the modified "non-proper" diving heuristic.

Conclusion

In this thesis, we have studied an industrial glass cutting problem, denoted as a two-dimensional bin-packing problem with leftovers, precedence constraints between bins and defects on bins ($2BP_{dpl}$). The main difficulties were to deal with defects on bins, industrial constraints and large instance size. We have proposed approaches based on column generation to solve the problem without defects consideration, reducing the problem to a two-dimensional bin-packing problem with leftovers ($2BP_l$). However a bottleneck of a direct column generation approach is the pricing subproblem, a two-dimensional 4-stage restricted exact guillotine knapsack problem (C-2KP-RE-4-r). In this document, we have focused on three different problems: the full pricing problem 2KP, which is solved by iterative dynamic programs, the $2BP_l$ without defects, and the $2BP_{dpl}$ with defects on bins, which are solved by diving heuristics.

The C-2KP-RE-4-r remains a challenge for exact methods when large size of instances are considered. For this problem, we have designed methods based on a network-flow formulation in hypergraphs, which is obtained by rewriting the dynamic program representing the set of all cutting patterns allowing item overproduction. By adding side-constraints to limit item production to this flow formulation, the C-2KP-RE-4-r can be solved with a generic MILP solver. However a limitation occurs quickly when large scale problem instances are considered. We use a more efficient method. Our original approach combines Lagrangian filtering, DSSR strategy and labelling algorithms on hypergraphs. To reduce hypergraph size and speed up algorithms, preprocessing techniques based on simple observations from cutting restrictions and partial pattern enumerations are also developed. Computational experiments demonstrate the efficiency of exact labelling methods.

New methods are designed to tackle the $2BP_l$ from results of 2KP solving. The main one is a so-called "non-proper" diving heuristic. From preliminary experimental computation, it was observed that solving a relaxed pricing problem with dynamic programming did not weaken too much the linear relaxation of the $2BP_l$ compared to solving the bounded knapsack problem. It is also faster to solve the dynamic program than solving the pricing problem with item bounds. The main drawback of this diving heuristic is that "non-proper" columns are added into the master problem. They are considered as unfeasible since they imply to cut multiple times an item whereas $2BP_l$ does not allow

item overproduction by definition. In order to obtain an integer solution, a described "non-proper" diving heuristic is used after column generation convergence. It iteratively fixes a proper column at each iteration. When the master problem only contains non-proper columns, a heuristic solution is built taking into account the dual costs. To keep short computation time efficiency of dynamic program, it is used to price columns after each variable fixing. Thus new found columns may be also "non-proper". Computational experiments outline the pertinence of this diving heuristic both in terms of solution quality and computation time.

From results obtained for the $2BP_l$, the industrial $2BP_{dpl}$, subject of this thesis, is then solved in two ways. First, the relaxed defect-free problem, *i.e.* $2BP_l$, is tackled with the "non-proper" diving heuristic and then different post-processing methods are used to obtain a feasible solution to the $2BP_{dpl}$. Most of post-processing methods use the structure of cutting patterns and mainly the guillotine cut property. However this approach is limited in term of solution quality since it only relies of a given $2BP_l$ solution. To avoid this behavior, the way to fix columns in the "non-proper" diving heuristic is modified to handle directly bin with defects. Each time a column is fixed, the bin to cut is also selected. Computational experiments are performed on different industrial instances with different defect distributions per bin. Results outline the good results of the "non-proper" diving heuristic handling bin with defects when columns are fixed.

To conclude, we proposed advanced techniques based on column generation and dynamic programming for solving heuristically our industrial glass cutting problem. The gap between the dual and primal bounds obtained is small, but the state-of-the-art methods do not allow to solve exactly real-size problems with defects.

From an industrial point of view, several variants of the problem could be addressed, including inventory constraints, time windows for orders, or optimizing the conception of batches.

Solving exactly the problem is a hard scientific challenge. Although it is easy to extend theoretically the dynamic program to account for defects, the size of the hypergraphs produced (one per different plate) would be huge. Even an approach based on relaxing item bounds would lead to models that would be too large for a regular computer memory. Several ideas can be used. The first one is to dynamically generate dynamic programs. The idea is first to work on a small but relaxed dynamic program and then extend it when needed. The difficulty of this approach is to solve the separation problem and to avoid a too fast expansion of the network created. If one is able to produce effective results using this approach, the "non-proper" diving heuristic can be used directly without modifying the way to fix columns. Since pricing problems are represented as flow problem in hypergraphs, it should also be possible to combine branch-and-price and consecutive branching on hyperarcs. It aims

to fix consecutive arc variables in pricing problems and is used to explore the branching tree. However extend consecutive branching to hypergraphs is not straightforward.

Acknowledgments

Experiments presented in this document were carried out using the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (see `https://www.plafrim.fr/`).

Bibliography

[1] Afsharian, M., Niknejad, A., Wäscher, G., 2014. A heuristic, dynamic programming-based approach for a two-dimensional cutting problem with defects. OR Spectrum 36 (4), 971–999.
URL http://dx.doi.org/10.1007/s00291-014-0363-x

[2] Alvarez-Valdés, R., Parajón, A., Tamarit, J. M., 2002. A tabu search algorithm for large-scale guillotine (un)constrained two-dimensional cutting problems. Computers & Operations Research 29 (7), 925 – 947.
URL http://dx.doi.org/10.1016/S0305-0548(00)00095-2

[3] Alvarez-Valdes, R., Parajon, A., Tamarit, M. J., 2002. A computational study of lp-based heuristic algorithms for two-dimensional guillotine cutting stock problems. OR Spectrum 24 (2), 179–192.
URL http://dx.doi.org/10.1007/s00291-002-0093-3

[4] Andrade, R., Birgin, E., Morabito, R., 2016. Two-stage two-dimensional guillotine cutting stock problems with usable leftover. International Transactions in Operational Research 23 (1-2), 121–145.
URL http://dx.doi.org/10.1111/itor.12077

[5] Beasley, J. E., 1985. Algorithms for unconstrained two-dimensional guillotine cutting. Journal of the Operational Research Society 36 (4), 297–306.
URL http://dx.doi.org/10.1057/jors.1985.51

[6] Berge, C., 1973. Graphs and Hypergraphs. North-Holland Mathematical Library. Amsterdam.

[7] Berkey, J. O., Wang, P. Y., 1987. Two-dimensional finite bin-packing algorithms. Journal of the Operational Research Society 38 (5), 423–429.
URL http://dx.doi.org/10.1057/jors.1987.70

[8] Birgin, E., Romao, O., Ronconi, D., 2018. The multiperiod two-dimensional non-guillotine cutting stock problem with usable leftovers. Tech. rep.
URL https://www.ime.usp.br/~egbirgin/publications/bromro.pdf

[9] Christofides, N., Hadjiconstantinou, E., 1995. An exact algorithm for orthogonal 2-D cutting problems using guillotine cuts. European Journal of Operational Research 83 (1), 21 – 38.
URL http://dx.doi.org/10.1016/0377-2217(93)E0277-5

[10] Christofides, N., Whitlock, C., 1977. An algorithm for two-dimensional cutting problems. Operations Research 25 (1), 30–44.
URL http://dx.doi.org/10.1287/opre.25.1.30

[11] Cintra, G., Miyazawa, F., Wakabayashi, Y., Xavier, E., 2008. Algorithms for two-dimensional cutting stock and strip packing problems using dynamic programming and column generation. European Journal of Operational Research 191 (1), 61 – 85.
URL http://dx.doi.org/10.1016/j.ejor.2007.08.007

[12] Clautiaux, F., Jouglet, A., Moukrim, A., 2013. A new graph-theoretical model for the guillotine-cutting problem. INFORMS Journal on Computing 25 (1), 72–86.
URL https://doi.org/10.1287/ijoc.1110.0478

[13] Clautiaux, F., Sadykov, R., Vanderbeck, F., Viaud, Q., 2018. Combining dynamic programming with filtering to solve a four-stage two-dimensional guillotine-cut bounded knapsack problem. Discrete Optimization.
URL https://doi.org/10.1016/j.disopt.2018.02.003

[14] Côté, J.-F., Iori, M., 2016. The meet-in-the-middle principle for cutting and packing problems. Tech. Rep. CIRRELT-2016-28, Montréal, Canada.

[15] Cui, Y.-P., Cui, Y., Tang, T., 2015. Sequential heuristic for the two-dimensional bin-packing problem. European Journal of Operational Research 240 (1), 43 – 53.
URL http://dx.doi.org/10.1016/j.ejor.2014.06.032

[16] Cui, Y.-P., Cui, Y., Tang, T., Hu, W., 2015. Heuristic for constrained two-dimensional three-staged patterns. Journal of the Operational Research Society 66 (4), 647–656.
URL http://dx.doi.org/10.1057/jors.2014.33

[17] Cung, V.-D., Hifi, M., Le Cun, B., 2000. Constrained two-dimensional cutting stock problems a best-first branch-and-bound algorithm. International Transactions in Operational Research 7 (3), 185–210.
URL http://dx.doi.org/10.1111/j.1475-3995.2000.tb00194.x

[18] Dantzig, G. B., Wolfe, P., 1960. Decomposition principle for linear programs. Operations Research 8 (1), 101–111.
URL https://doi.org/10.1287/opre.8.1.101

[19] De Gelder, E. R., Wagelmans, A. P. M., 2009. The two-dimensional cutting stock problem within the roller blind production process. Statistica Neerlandica 63 (4), 474–489.
URL http://dx.doi.org/10.1111/j.1467-9574.2009.00436.x

[20] Delorme, M., Iori, M., Martello, S., 2016. Bin packing and cutting stock problems: Mathematical models and exact algorithms. European Journal of Operational Research 255 (1), 1 – 20.
URL https://doi.org/10.1016/j.ejor.2016.04.030

[21] Desaulniers, G., Desrosiers, J., Solomon, M. M., 2006. Column generation. Vol. 5. Springer Science & Business Media.

[22] Detienne, B., Sadykov, R., Tanaka, S., 2016. The two-machine flow-shop total completion time problem: Branch-and-bound algorithms based on network-flow formulation. European Journal of Operational Research 252 (3), 750 – 760.
URL http://dx.doi.org/10.1016/j.ejor.2016.02.003

[23] Dolatabadi, M., Lodi, A., Monaci, M., 2012. Exact algorithms for the two-dimensional guillotine knapsack. Computers & Operations Research 39 (1), 48–53.
URL http://dx.doi.org/10.1016/j.cor.2010.12.018

[24] Dusberger, F., Raidl, G. R., 2014. A variable neighborhood search using very large neighborhood structures for the 3-staged 2-dimensional cutting stock problem. In: Blesa, M. J., Blum, C., Voß, S. (Eds.), Hybrid Metaheuristics: 9th International Workshop, HM 2014, Hamburg, Germany, June 11-13, 2014. Proceedings. Springer International Publishing, Cham, pp. 85–99.
URL http://dx.doi.org/10.1007/978-3-319-07644-7_7

[25] Dusberger, F., Raidl, G. R., 2015. Solving the 3-staged 2-dimensional cutting stock problem by dynamic programming and variable neighborhood search. Electronic Notes in Discrete Mathematics 47, 133 – 140.
URL http://dx.doi.org/10.1016/j.endm.2014.11.018

[26] Edmonds, J., 1965. Paths, trees, and flowers. Canadian Journal of mathematics 17 (3), 449–467.
URL http://dx.doi.org/10.4153/CJM-1965-045-4

[27] Fayard, D., Hifi, M., Zissimopoulos, V., Dec 1998. An efficient approach for large-scale two-dimensional guillotine cutting stock problems. Journal of the Operational Research Society 49 (12), 1270–1277.
URL https://doi.org/10.1057/palgrave.jors.2600638

[28] Fekete, S., Schepers, J., 1997. A new exact algorithm for general orthogonal d-dimensional knapsack problems. In: Burkard, R., Woeginger, G. (Eds.), Algorithms — ESA '97. Vol. 1284 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 144–156.
URL http://dx.doi.org/10.1007/3-540-63397-9_12

[29] Furini, F., Malaguti, E., Durán, R. M., Persiani, A., Toth, P., 2012. A column generation heuristic for the two-dimensional two-staged guillotine cutting stock problem with multiple stock size. European Journal of Operational Research 218 (1), 251 – 260.
URL http://dx.doi.org/10.1016/j.ejor.2011.10.018

[30] Furini, F., Malaguti, E., Thomopulos, D., 2016. Modeling two-dimensional guillotine cutting problems via integer programming. INFORMS Journal on Computing 28 (4), 736–751.
URL http://dx.doi.org/10.1287/ijoc.2016.0710

[31] G, Y.-G., Seong, Y.-J., Kang, M.-K., 2003. A best-first branch and bound algorithm for unconstrained two-dimensional cutting problems. Operations Research Letters 31 (4), 301 – 307.
URL http://dx.doi.org/10.1016/S0167-6377(03)00002-6

[32] Gallo, G., Longo, G., Pallottino, S., Nguyen, S., 1993. Directed hypergraphs and applications. Discrete Applied Mathematics 42 (2–3), 177 – 201.
URL http://dx.doi.org/10.1016/0166-218X(93)90045-P

[33] Gilmore, P. C., Gomory, R. E., 1961. A linear programming approach to the cutting-stock problem. Operations research 9 (6), 849–859.
URL http://dx.doi.org/10.1287/opre.9.6.849

[34] Gilmore, P. C., Gomory, R. E., 1965. Multistage cutting stock problems of two and more dimensions. Operations research 13 (1), 94–120.
URL http://dx.doi.org/10.1287/opre.13.1.94

[35] Hadjiconstantinou, E., Iori, M., 2007. A hybrid genetic algorithm for the two-dimensional single large object placement problem. European Journal of Operational Research 183 (3), 1150 – 1166.
URL http://dx.doi.org/10.1016/j.ejor.2005.11.061

[36] Hahn, S. G., 1968. On the optimal cutting of defective sheets. Operations Research 16 (6), 1100–1114.
URL http://dx.doi.org/10.1287/opre.16.6.1100

[37] Held, M., Wolfe, P., Crowder, H. P., 1974. Validation of subgradient optimization. Mathematical programming 6 (1), 62–88.
URL http://dx.doi.org/10.1007/BF01580223

[38] Herz, J. C., Sept 1972. Recursive computational procedure for two-dimensional stock cutting. IBM Journal of Research and Development 16 (5), 462–469.
URL http://dx.doi.org/10.1147/rd.165.0462

[39] Hifi, M., 1997. An improvement of viswanathan and bagchi's exact algorithm for constrained two-dimensional cutting stock. Computers & Operations Research 24 (8), 727 – 736.
URL http://dx.doi.org/10.1016/S0305-0548(96)00095-0

[40] Hifi, M., 2001. Exact algorithms for large-scale unconstrained two and three staged cutting problems. Computational Optimization and Applications 18 (1), 63–88.
URL http://dx.doi.org/10.1023/A:1008743711658

[41] Hifi, M., Mar 2004. Dynamic programming and hill-climbing techniques for constrained two-dimensional cutting stock problems. Journal of Combinatorial Optimization 8 (1), 65–84.
URL https://doi.org/10.1023/B:JOCO.0000021938.49750.91

[42] Hifi, M., Roucairol, C., 2001. Approximate and exact algorithms for constrained (un) weighted two-dimensional two-staged cutting stock problems. Journal of Combinatorial Optimization 5 (4), 465–494.
URL http://dx.doi.org/10.1023/A:1011628809603

[43] Ibaraki, T., Nakamura, Y., 1994. A dynamic programming method for single machine scheduling. European Journal of Operational Research 76 (1), 72 – 82.
URL http://dx.doi.org/10.1016/0377-2217(94)90007-8

[44] Irnich, S., Desaulniers, G., Desrosiers, J., Hadjar, A., 2010. Path-reduced costs for eliminating arcs in routing and scheduling. INFORMS Journal on Computing 22 (2), 297–313.
URL http://dx.doi.org/10.1287/ijoc.1090.0341

[45] Jin, M., Ge, P., Ren, P., 2015. A new heuristic algorithm for two-dimensional defective stock guillotine cutting stock problem with multiple stock sizes. Tehnički vjesnik 22 (5), 1107–1116.
URL http://dx.doi.org/10.17559/TV-20150731113849

[46] Kantorovich, L. V., 1960. Mathematical methods of organizing and planning production. Management Science 6 (4), 366–422.
URL https://doi.org/10.1287/mnsc.6.4.366

[47] Lodi, A., Martello, S., Vigo, D., 1999. Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. INFORMS

Journal on Computing 11 (4), 345–357.
URL http://dx.doi.org/10.1287/ijoc.11.4.345

[48] Lodi, A., Martello, S., Vigo, D., 2002. Recent advances on two-dimensional bin packing problems. Discrete Applied Mathematics 123 (1–3), 379 – 396.
URL http://dx.doi.org/10.1016/S0166-218X(01)00347-X

[49] Lodi, A., Martello, S., Vigo, D., 2004. Models and bounds for two-dimensional level packing problems. Journal of Combinatorial Optimization 8 (3), 363–379.
URL http://dx.doi.org/10.1023/B:JOCO.0000038915.62826.79

[50] Lodi, A., Monaci, M., 2003. Integer linear programming models for 2-staged two-dimensional knapsack problems. Mathematical Programming 94 (2-3), 257–278.
URL http://dx.doi.org/10.1007/s10107-002-0319-9

[51] Lodi, A., Monaci, M., Pietrobuoni, E., 2017. Partial enumeration algorithms for two-dimensional bin packing problem with guillotine constraints. Discrete Applied Mathematics 217, 40 – 47.
URL https://doi.org/10.1016/j.dam.2015.09.012

[52] Macedo, R., Alves, C., Valério de Carvalho, J. M., 2010. Arc-flow model for the two-dimensional guillotine cutting stock problem. Computers & Operations Research 37 (6), 991–1001.
URL http://dx.doi.org/10.1016/j.cor.2009.08.005

[53] Martello, S., Pisinger, D., Toth, P., 1999. Dynamic programming and strong bounds for the 0-1 knapsack problem. Management Science 45 (3), 414–424.
URL https://doi.org/10.1287/mnsc.45.3.414

[54] Martello, S., Toth, P., 1990. Knapsack Problems: Algorithms and Computer Implementations. John Wiley & Sons, Inc., New York, NY, USA.

[55] Martello, S., Vigo, D., 1998. Exact solution of the two-dimensional finite bin packing problem. Management Science 44 (3), 388–399.
URL http://dx.doi.org/10.1287/mnsc.44.3.388

[56] Martin, R. K., Rardin, R. L., Campbell, B. A., 1990. Polyhedral characterization of discrete dynamic programming. Operations Research 38 (1), 127–138.
URL http://dx.doi.org/10.1287/opre.38.1.127

[57] Martinelli, R., Pecin, D., Poggi, M., 2014. Efficient elementary and restricted non-elementary route pricing. European Journal of Operational Research 239 (1), 102 – 111.
URL http://dx.doi.org/10.1016/j.ejor.2014.05.005

[58] Morabito, R., Arenales, M. N., 1996. Staged and constrained two-dimensional guillotine cutting problems: An and/or-graph approach. European Journal of Operational Research 94 (3), 548–560.
URL http://dx.doi.org/10.1016/0377-2217(95)00128-X

[59] Morabito, R., Pureza, V., 2010. A heuristic approach based on dynamic programming and and/or-graph search for the constrained two-dimensional guillotine cutting problem. Annals of Operations Research 179 (1), 297–315.
URL http://dx.doi.org/10.1007/s10479-008-0457-4

[60] Neidlein, V., Vianna, A. C., Arenales, M. N., Wäscher, G., 2009. Two-Dimensional Guillotineable-Layout Cutting Problems with a Single Defect - An AND/OR-Graph Approach. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 85–90.
URL http://dx.doi.org/10.1007/978-3-642-00142-0_14

[61] Nemhauser, G., Wolsey, L., 1988. Integer and Combinatorial Optimization. Wiley.
URL http://dx.doi.org/10.1002/9781118627372

[62] Oliveira, J. F., Ferreira, J. S., 1990. An improved version of Wang's algorithm for two-dimensional cutting problems. European Journal of Operational Research 44 (2), 256 – 266.
URL http://dx.doi.org/10.1016/0377-2217(90)90361-E

[63] Ozdamar, L., 2000. The cutting-wrapping problem in the textile industry: optimal overlap of fabric lengths and defects for maximizing return based on quality. International Journal of Production Research 38 (6), 1287–1309.
URL http://dx.doi.org/10.1080/002075400188852

[64] Pisinger, D., 1995. An expanding-core algorithm for the exact 0–1 knapsack problem. European Journal of Operational Research 87 (1), 175 – 187.
URL https://doi.org/10.1016/0377-2217(94)00013-3

[65] Polyakovsky, S., M'Hallah, R., 2009. An agent-based approach to the two-dimensional guillotine bin packing problem. European Journal of Operational Research 192 (3), 767 – 781.
URL http://dx.doi.org/10.1016/j.ejor.2007.10.020

[66] Puchinger, J., Raidl, G. R., 2007. Models and algorithms for three-stage two-dimensional bin packing. European Journal of Operational Research 183 (3), 1304–1327.
URL http://dx.doi.org/10.1016/j.ejor.2005.11.064

[67] Puchinger, J., Raidl, G. R., Koller, G., 2004. Solving a real-world glass cutting problem. In: Gottlieb, J., Raidl, G. R. (Eds.), Evolutionary Computation in Combinatorial Optimization: 4th European Conference, EvoCOP 2004, Coimbra, Portugal, April 5-7, 2004. Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 165–176.
URL http://dx.doi.org/10.1007/978-3-540-24652-7_17

[68] Righini, G., Salani, M., 2008. New dynamic programming algorithms for the resource constrained elementary shortest path problem. Networks 51 (3), 155–170.
URL http://dx.doi.org/10.1002/net.v51:3

[69] Rönnqvist, M., Åstrand, E., 1998. Integrated defect detection and optimization for cross cutting of wooden boards. European Journal of Operational Research 108 (3), 490 – 508.
URL http://dx.doi.org/10.1016/S0377-2217(97)00181-1

[70] Russo, M., Sforza, A., Sterle, C., 2014. An exact dynamic programming algorithm for large-scale unconstrained two-dimensional guillotine cutting problems. Computers & Operations Research 50, 97 – 114.
URL http://dx.doi.org/10.1016/j.cor.2014.04.001

[71] Sadykov, R., Vanderbeck, F., 2013. Column generation for extended formulations. EURO Journal on Computational Optimization 1 (1-2), 81–115.
URL http://dx.doi.org/10.1007/s13675-013-0009-9

[72] Sadykov, R., Vanderbeck, F., Pessoa, A., Tahiri, I., Uchoa, E., 2016. Primal heuristics for branch-and-price: the assets of diving methods. Tech. Rep. hal-01237204, HAL Inria.
URL https://hal.inria.fr/hal-01237204/

[73] Sarker, B. R., 1988. An optimum solution for one-dimensional slitting problems: A dynamic programming approach. Journal of the Operational Research Society 39 (8), 749–755.
URL http://dx.doi.org/10.1057/jors.1988.130

[74] Scheithauer, G., Terno, J., 1988. Guillotine cutting of defective boards. Optimization 19 (1), 111–121.
URL http://dx.doi.org/10.1080/02331938808843323

[75] Silva, E., Alvelos, F., Valério de Carvalho, J. M., 2010. An integer programming model for two-and three-stage two-dimensional cutting stock problems. European Journal of Operational Research 205 (3), 699–708.
URL http://dx.doi.org/10.1016/j.ejor.2010.01.039

[76] Tomat, L., Gradisar, M., 2017. One-dimensional stock cutting: optimization of usable leftovers in consecutive orders. Central European Journal of Operations Research 25 (2), 473–489.
URL https://doi.org/10.1007/s10100-017-0466-y

[77] Trkman, P., Gradisar, M., 2007. One-dimensional cutting stock optimization in consecutive time periods. European Journal of Operational Research 179 (2), 291 – 301.
URL https://doi.org/10.1016/j.ejor.2006.03.027

[78] Twisselmann, U., 1999. Cutting rectangles avoiding rectangular defects. Applied Mathematics Letters 12 (6), 135 – 138.
URL http://dx.doi.org/10.1016/S0893-9659(99)00092-0

[79] Valério de Carvalho, J. M., 1999. Exact solution of bin-packing problems using column generation and branch-and-bound. Annals of Operations Research 86, 629–659.
URL http://dx.doi.org/10.1023/A:1018952112615

[80] Valério de Carvalho, J. M., 2002. Lp models for bin packing and cutting stock problems. European Journal of Operational Research 141 (2), 253 – 273.
URL http://dx.doi.org/10.1016/S0377-2217(02)00124-8

[81] Vance, P. H., 1998. Branch-and-price algorithms for the one-dimensional cutting stock problem. Computational Optimization and Applications 9 (3), 211–228.
URL https://doi.org/10.1023/A:1018346107246

[82] Vanderbeck, F., 1999. Computational study of a column generation algorithm for bin packing and cutting stock problems. Mathematical Programming 86 (3), 565–594.
URL https://doi.org/10.1007/s101070050105

[83] Vanderbeck, F., 2001. A nested decomposition approach to a three-stage, two-dimensional cutting-stock problem. Management Science 47 (6), 864–879.
URL http://dx.doi.org/10.1287/mnsc.47.6.864.9809

[84] Vanderbeck, F., 2011. Branching in branch-and-price: a generic scheme. Mathematical Programming 130 (2), 249–294.
URL https://doi.org/10.1007/s10107-009-0334-1

[85] Velasco, A., Uchoa, E., 2017. Improved space-state relaxation for constrained two-dimensional guillotine cutting problems. Tech. Rep. L-2017-1, Cadernos do LOGIS-UFF, Niterói, Brazil.

[86] Wang, P. Y., 1983. Two algorithms for constrained two-dimensional cutting stock problems. Operations Research 31 (3), 573–586.
URL http://dx.doi.org/10.1287/opre.31.3.573

[87] Wäscher, G., Haußner, H., Schumann, H., 2007. An improved typology of cutting and packing problems. European Journal of Operational Research 183 (3), 1109 – 1130.
URL http://dx.doi.org/10.1016/j.ejor.2005.12.047