Instituto de Computação
Universidade Estadual de Campinas

Algoritmos para Problemas de Corte e Empacotamento

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Thiago Alves de Queiroz e aprovada pela Banca Examinadora.

Campinas, 21 de dezembro de 2010.

Prof. Dr. Flávio Keidi Miyazawa (Orientador)

Tese apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Fabiana Bezerra Müller – CRB8 / 6162

Queiroz, Thiago Alves de
Q32a Algoritmos para problemas de corte e empacotamento/Thiago Alves de Queiroz-- Campinas, [S.P. : s.n.], 2010.

Orientador : Flávio Keidi Miyazawa.
Tese (doutorado) - Universidade Estadual de Campinas, Instituto de Computação.

1.Problema de corte. 2.Problema de empacotamento. 3.Otimização combinatória. 4.Algoritmos. I. Miyazawa, Flávio Keidi. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Algorithms for cutting and packing problems

Palavras-chave em inglês (Keywords): 1. Cutting problem. 2. Packing problem. 3. Combinatorial optimization. 4. Algorithms.

Titulação: Doutor em Ciência da Computação

Banca examinadora: Prof. Dr. Flávio Keidi Miyazawa (IC – UNICAMP)
Prof. Dr. Eduardo Cândido Xavier (IC – UNICAMP)
Prof. Dr. Glauber Ferreira Cintra (DI – IFCE)
Prof. Dr. Horácio Hideki Yanasse (CTE – INPE)
Prof. Dr. Orlando Lee (IC - UNICAMP)

Data da defesa: 21/12/2010

Programa de Pós-Graduação: Doutorado em Ciência da Computação

TERMO DE APROVAÇÃO

Tese Defendida e Aprovada em 21 de dezembro de 2010, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Horacio Hideki Yanasse
INPE

Prof. Dr. Glauber Ferreira Cintra
Departamento de Informática / IFCE

Prof. Dr. Eduardo Candido Xavier
IC / UNICAMP

Prof. Dr. Orlando Lee
IC / UNICAMP

Prof. Dr. Flávio Keidi Miyazawa
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Algoritmos para Problemas de Corte e Empacotamento

Thiago Alves de Queiroz[1]

Dezembro de 2010

Banca Examinadora:

- Prof. Dr. Flávio Keidi Miyazawa (Orientador)
- Prof. Dr. Eduardo Cândido Xavier
  Instituto de Computação, UNICAMP
- Prof. Dr. Glauber Ferreira Cintra
  Departamento de Informática, IFCE
- Prof. Dr. Horácio Hideki Yanasse
  Centro de Tecnologias Especiais, INPE
- Prof. Dr. Orlando Lee
  Instituto de Computação, UNICAMP

[1]Pesquisa financiada em partes pela CAPES, CNPq (processo 140834/2009-5) e FAPESP (processo 2009/01129-1).

Resumo

Problemas de Corte e Empacotamento são, em sua maioria, $\mathcal{NP}$-difíceis e não existem algoritmos exatos de tempo polinomial para tais se for considerado $P \neq NP$. Aplicações práticas envolvendo estes problemas incluem a alocação de recursos para computadores; o corte de chapas de ferro, de madeira, de vidro, de alumínio, peças em couro, etc.; a estocagem de objetos; e, o carregamento de objetos dentro de contêineres ou caminhões-baú.

Nesta tese investigamos problemas de Corte e Empacotamento $\mathcal{NP}$-difíceis, nas suas versões bi- e tridimensionais, considerando diversas restrições práticas impostas a tais, a saber: que permitem a rotação ortogonal dos itens; cujos cortes sejam feitos por uma guilhotina; cujos cortes sejam feitos por uma guilhotina respeitando um número máximo de estágios de corte; cujos cortes sejam não-guilhotinados; cujos itens tenham demanda (não) unitária; cujos recipientes tenham tamanhos diferentes; cujos itens sejam representados por polígonos convexos e não-convexos (formas irregulares); cujo empacotamento respeite critérios de estabilidade para corpos rígidos; cujo empacotamento satisfaça uma dada ordem de descarregamento; e, cujos empacotamentos intermediários e final tenham seu centro de gravidade dentro de uma região considerada "segura".

Para estes problemas foram propostos algoritmos baseados em programação dinâmica; modelos de programação inteira; técnicas do tipo *branch-and-cut*; heurísticas, incluindo as baseadas na técnica de geração de colunas; e, meta-heurísticas como o GRASP. Resultados teóricos também foram obtidos. Provamos uma questão em aberto levantada na literatura sobre cortes não-guilhotinados restritos a um conjunto de pontos.

Uma extensiva série de testes computacionais considerando instâncias reais e várias outras geradas de forma aleatória foram realizados com os algoritmos desenvolvidos. Os resultados computacionais, sendo alguns deles comparados com a literatura, comprovam a validade das algoritmos propostos e a sua aplicabilidade prática para resolver os problemas investigados.

Abstract

Several versions of Cutting and Packing problems are considered $\mathcal{NP}$-hard and, if we consider that $P \neq NP$, we do not have any exact polynomial algorithm for solve them. Practical applications arises for such problems and include: resources allocation for computers; cut of steel, wood, glass, aluminum, etc.; packing of objects; and, loading objects into containers and trucks.

In this thesis we investigate Cutting and Packing problems that are $\mathcal{NP}$-hard considering theirs two- and three-dimensional versions, and subject to several practical constraints, that are: that allows the items to be orthogonally rotated; whose cuts are guillotine type; whose cuts are guillotine type and performed in at most k stages; whose cuts are non-guillotine type; whose items have varying and unit demand; whose bins are of variable sizes; whose items are represented by convex and non-convex polygons (irregular shapes); whose packing must satisfy the conditions for static equilibrium of rigid bodies; whose packing must satisfy an order to unloading; and, whose intermediaries and resultant packing have theirs center of gravity inside a safety region;

Such cutting and packing problems were solved by dynamic programming algorithms; integer linear programming models; *branch-and-cut* algorithms; several heuristics, including those ones based on column generation approaches, and metaheuristics like GRASP. Theoretical results were also provided, so a recent open question arised by literature about non-guillotine patterns restricted to a set of points was demonstrated.

We performed an extensive series of computational experiments for algorithms developed considering several instances presented in literature and others generated at random. These results, some of them compared with the literature, validate the approaches proposed and suggest their applicability to deal with practical situations involving the problems here investigated.

Agradecimentos

Agradeço...

... a Deus pelo pão de todos os dias: *"...ainda que eu andasse pelo vale da sombra da morte, não temeria mal algum..."*, *"...porque ele te livrará do laço do passarinheiro, e da peste perniciosa..."*.

.... à minha família, principalmente meus pais, Orlando e Charlene, e minhas irmãs, Joyce e Monique, por todo o apoio, carinho e sermões.

.... à minha noiva Layane por seu amor verdadeiro e por estar ao meu lado em todos os momentos.

.... ao meu orientador, grande amigo Flávio K. Miyazawa, por me aconselhar e orientar tanto nos estudos como na vida.

... aos amigos, principalmente André, Lehilton e Thiago de Paulo, pelas boas conversas paralelas e horas de AGE.

... aos colegas (alunos, professores e técnicos) do IC/UNICAMP que me ajudaram, mesmo que implicitamente, nesta jornada.

... a todos aqueles que contribuíram, com boa ou má fé, para a realização deste trabalho.

... a CAPES, CNPq e FAPESP, pelo apoio financeiro.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Atualmente existe uma grande demanda por recursos oriundos de diversas fontes, seja recursos para a indústria ou pequenas empresas, até mesmo para usuários domésticos. Estes recursos, estando armazenados em algum depósito, precisam ser alocados em meios de transporte para, então, serem entregues aos destinatários finais.

Segundo estudos, veja [62], o valor final das mercadorias está acrescido de um percentual de 10% a 15% devido aos gastos com o processo de logística, que incluem, por exemplo, a estocagem, o carregamento e o eventual transporte das mercadorias. No contexto nacional, pesquisas mostram que estes serviços representam em média 2,53% do PIB (produto interno bruto) de um país.

Nesta perspectiva, muitas vezes, os itens estocados precisam ser cortados em pedaços menores, caracterizando um problema de corte (*Cutting Problem*). Por outro lado, tais itens precisam ser carregados (empacotados) em objetos maiores, recipientes, caracterizando um problema de empacotamento (*Packing Problem*). Em ambas as terminologias, estamos interessados em otimizar o trabalho efetuado, seja minimizando o desperdício de matéria-prima, no caso dos problemas de corte, seja maximizando o uso do recipiente, nos problemas de empacotamento.

Apesar da diferença nos termos corte e empacotamento, estes problemas apresentam uma estrutura lógica análoga para fins teóricos. Isto possibilita tratar um problema de corte como sendo de empacotamento e vice-versa. Note que cortar um recipiente para obter um conjunto de itens equivale a empacotar o conjunto de itens dentro do recipiente. Deste modo, usaremos o termo corte ou empacotamento sem fazer distinção entre tais, referenciando os problemas como *Problemas de Corte e Empacotamento*.

Organizar itens dentro de recipientes é uma tarefa complexa e importante. Vale destacar que ainda existem empresas que trabalham com métodos manuais de empacotamento, o que resulta, muitas vezes, no uso ineficiente dos recipientes. Além disso, fatores importantes durante a organização podem ser considerados, como a estabilidade e a fragilidade dos itens, a estabilidade do empacotamento final, satisfazer uma determinada ordem de descarregamento dos itens (itens

mais pesados devem ficar embaixo, por exemplo), entre outros.

Nota-se, então, que este não é um problema simples, pois não envolve apenas empacotar/cortar (de qualquer maneira) itens dentro de recipientes. Torna-se necessário diminuir os custos envolvidos, respeitando diversas restrições impostas e algumas até obrigatórias devido às questões de segurança.

O objetivo deste trabalho está em propor algoritmos eficientes para alguns problemas de corte e empacotamento, incluindo aqueles que devem satisfazer restrições presentes no mundo real. O termo "eficiente" é usado no sentido de que estamos interessados em abordagens que gerem soluções ótimas ou próximas delas dentro de um tempo de processamento aceitável, já que este tempo pode ser decisivo em problemas práticos como os tratados nesta tese. Para tanto, os algoritmos propostos incluem abordagens exatas e heurísticas.

1.1 Motivação

Os problemas de Corte e Empacotamento são fáceis de serem entendidos. Porém, do ponto de vista computacional, são bastante complexos. Em linhas gerais, grande parte destes problemas não possuem algoritmos exatos de tempo polinomial, supondo $P \neq \mathcal{NP}$. Tais problemas vêm sendo investigados desde os anos 60 [9, 11, 17, 20, 27, 41, 71]. Muitas de suas variantes são $\mathcal{NP}$-difíceis, precisamente aquelas apresentadas neste trabalho.

Diversas são as aplicações práticas que envolvem problemas desta natureza, como na alocação de recursos para computadores; no corte de tecido, de bobinas de alumínio, de madeira, chapas de ferro, lâminas de vidro, peças em couro; etc. Algumas indústrias de bens de consumo precisam empacotar seus produtos e estocá-los em armazéns. Empresas de transporte necessitam carregar tais produtos dentro de contêineres ou caminhões-baú, podendo ainda, estes contêineres serem organizados/transportados dentro de navios. Em relação ao carregamento de contêineres, estes, em geral, possuem dimensões fixas e padronizadas. Tal padronização permite que eles sejam usados tanto por equipamentos de carga/descarga como no acoplamento a caminhões. Assim, qualquer que seja a aplicação, o objetivo consiste em otimizar o processo de corte/empacotamento, visando minimizar custos e/ou maximizar lucros.

Diversas são as áreas do conhecimento que trabalham com problemas desta natureza, como a matemática, a engenharia, a economia e a computação. A resolução de tais podem envolver tanto o uso de abordagens exatas como heurísticas. Podemos destacar: *programação dinâmica*; *modelos de programação linear*; técnicas *branch-and-bound*, *branch-and-cut*, *branch-and-price*, *branch-and-cut-and-price*; *busca Tabu*, *algoritmos evolutivos*, *GRASP*, etc.

A diferença entre cada abordagem de resolução é crucial e depende da necessidade do usuário, pois o uso de abordagens exatas envolve um número exponencial de combinações, que cresce de acordo com o tamanho da instância e, em geral, é voltada para instâncias de pequeno e médio porte. O uso de heurísticas pode tratar instâncias maiores, porém podem levar

a soluções inviáveis na prática. É importante mencionar que abordagens heurísticas são extremamente importantes em abordagens exatas, pois na maioria das vezes, permitem produzir soluções viáveis e possibilitam diminuir o espaço de busca.

Ao propor novos algoritmos para problemas de corte e empacotamento, buscaremos realizar testes com várias instâncias do mundo real, além de instâncias geradas de forma aleatória, sempre com o intuito de comparar nossos resultados com resultados da literatura e mostrar a aplicabilidade prática das estratégias desenvolvidas. Também desejamos demonstrar que a metodologia seguida é bem fundamentada e realmente útil para ser aplicada na resolução real dos problemas citados e demais variantes.

1.2 Conceitos, Problemas e Técnicas

Nesta seção apresentaremos conceitos e informações básicas requeridas nos próximos capítulos. Inicialmente, conceitos sobre problemas de corte e empacotamento são apresentados. Depois, definimos os principais problemas estudados. Também relatamos brevemente as abordagens utilizadas para resolver tais problemas, além de outras questões pertinentes.

1.2.1 Alguns Conceitos

Por simplicidade, os objetos grandes são preferencialmente referenciados como recipientes, enquanto os pequenos por itens. Neste caso não fazemos distinção se o recipiente/item apresenta uma forma regular, por exemplo, é um retângulo ou um paralelepípedo, ou uma forma irregular, por exemplo, é um polígono qualquer (convexo ou não-convexo). Assim, quando o termo *irregular* aparecer após o objeto significa que estamos tratando de problemas com objetos irregulares. De outro modo, o objeto é considerado regular.

Como existe uma grande variedade de problemas de corte e empacotamento, estes podem ser classificados de acordo com certas características. Uma tipologia foi proposta por Wäscher *et al.* (2007) [86]. Ela parte da classificação proposta por Dyckhoff (1990) [33], fazendo algumas modificações, sendo mais precisa e cobrindo um maior número de problemas. Na tipologia de Wäscher *et al.* (2007), os problemas são classificados através de cinco critérios: dimensão, tipo de alocação, sortimento dos itens, sortimento dos recipientes, forma dos itens.

Um empacotamento viável requer que os itens sejam organizados de forma a não ocupar o mesmo local dentro do recipiente, ou seja, quaisquer dois itens dentro de um recipiente não podem se sobrepor, além disso não podem extrapolar as dimensões do recipiente. Chamamos de *padrão de corte* (ou *padrão*) cada possível maneira de se cortar um dado recipiente. Um *padrão homogêneo* ocorre quando os itens produzidos através deste padrão são de apenas um tipo. Os cortes são ortogonais quando os itens são obtidos através de cortes paralelos aos lados do recipiente.

Os cortes são classificados em guilhotinados e não-guilhotinados. Os cortes guilhotinados são efetuados por uma guilhotina. Neste caso, tais cortes vão (em linha reta) de um lado do recipiente até o lado oposto, sendo paralelos a dois lados. Denominamos de *padrão guilhotinado* um padrão obtido a partir de uma sequência de cortes de guilhotina que são aplicados ao recipiente original e aos subsequentes recipientes menores obtidos de cada corte (anterior). Um *padrão não-guilhotinado* é descrito por um padrão que não necessariamente é obtido através de cortes de guilhotina.

Nas aplicações práticas existe um limite quanto ao número de estágios de corte para obter os itens finais. Em um padrão guilhotinado chamamos de *estágio de corte* (ou *estágio*) uma sequência maximal de cortes consecutivos na mesma direção. Os cortes de um estágio devem ser perpendiculares aos cortes do estágio anterior. Além disso, o número de estágios mede a quantidade de mudanças na direção do corte. Deste modo, dizemos que um padrão guilhotinado é k*-estágios* se ele é obtido após executar k estágios de corte. Em alguns casos, torna-se necessário considerar um estágio adicional de corte que permite separar um item de uma parte não utilizável.

Uma outra característica dos problemas de corte e empacotamento envolve a orientação dos itens. No caso de *orientação fixa* os itens devem ser cortados/empacotados em suas dimensões originais, isto é, aquelas especificadas na instância de entrada. Por outro lado, quando os itens podem ser (ortogonalmente) rotacionados, a rotação ocorre em torno de um número limitado de eixos ou em torno de todos os eixos. E, ao rotacionar um item, o item resultante deve ser viável.

Uma outra restrição, menos considerada, mas muito importante na prática, diz respeito à estabilidade dos itens dentro do recipiente. Dizemos, informalmente, que um item está *estável* se, após tal item ser organizado dentro do recipiente, ele não venha a rotacionar e/ou desfazer o empacotamento feito. Neste caso, considerando apenas o efeito da força de gravidade. Para todo o conjunto de itens significa que, em qualquer instante, a forma como os itens estão organizados no recipiente não resulta no tombamento do empacotamento feito.

Outro caso que envolve a estabilidade consiste em manter o centro de gravidade do empacotamento o mais próximo possível de um centro de gravidade ideal, ou dentro de uma região considerada "segura". Deste modo, o recipiente não venha a inclinar (ou tombar) quando estiver sendo transportado, por exemplo.

Em situações reais, há também a necessidade dos itens serem empacotados segundo uma dada ordem que posteriormente favoreça o descarregamento. Por exemplo, para empresas que efetuam a entrega de mercadorias pode ser importante que os itens sejam organizados segundo a rota de entrega. Isto significa que clientes visitados primeiro, durante o descarregamento dos itens, tenham seus itens não obstruídos (impedidos de serem retirados) por itens de clientes que serão visitados posteriormente. Chamamos esta restrição de *restrição de ordem*.

Em linhas gerais, a restrição de ordem diz que itens com valor de ordem mais baixa (itens de clientes a serem visitados primeiro) não devem ser obstruídos por itens com ordem mais alta,

considerando a abertura no recipiente por onde tais itens serão descarregados.

As restrições discutidas acima contemplam as principais restrições que foram consideradas (de forma individual ou em grupo) quando resolvendo os problemas de corte e empacotamento nesta tese.

1.2.2 Problemas Base

Enunciaremos nesta subseção os principais problemas tratados nesta tese. Eles foram enunciados de forma geral, isto é, sem especificar a dimensão de interesse, e sem qualquer restrição adicional. Ao adicionar qualquer restrição, como aquelas discutidas na subseção anterior, um novo problema surge, mais especificamente, uma nova variante do problema surge.

PROBLEMA DA MOCHILA (*Knapsack Problem*): São dados um recipiente/mochila B e uma lista T de n (tipos de) itens i com valor v_i, para $i = 1, \ldots, n$. O objetivo é determinar como empacotar — num único recipiente B — um subconjunto de itens de maneira a maximizar o valor total dos itens empacotados.

A definição do problema da mochila acima corresponde a versão restrita, isto é, um item pode ser empacotado no máximo uma vez. Esta versão também é referenciada como mochila 0/1 (*0/1 Knapsack*). Por outro lado, na versão irrestrita (*Unconstrained Knapsack*) não há limite quanto ao número de cópias de cada item que pode ser empacotado.

PROBLEMA DO CORTE DE ESTOQUE (*Cutting Stock Problem*): São dados recipientes B, e uma lista T de n (tipos de) itens i com demanda d_i, para $i = 1, \ldots, n$. O objetivo é determinar como cortar o menor número possível de recipientes B para produzir d_i unidades de cada item i.

PROBLEMA DO EMPACOTAMENTO EM FAIXA (*Strip Packing Problem*): São dados uma faixa B, isto é, um recipiente com uma de suas dimensões aberta (sem um valor definido) e uma lista T de n (tipos de) itens i, para $i = 1, \ldots, n$. O objetivo consiste em decidir como cortar a faixa B de maneira que cada item i seja produzido e a dimensão, sem valor definido, da parte da faixa que é usada seja minimizada.

Se o valor da demanda (multiplicidade de cada item) for igual a um no problema do corte de estoque, surge, então, o problema *Bin Packing*. No problema de empacotamento em faixa também podemos associar uma demanda para os itens.

Diversas variantes dos problemas acima foram considerados nesta tese e são discutidas nos capítulos seguintes. Todas pertencem a classe $\mathcal{NP}$-difícil. De antemão, consideramos tanto as versões bidimensionais como as tridimensionais.

Nas versões bidimensionais estudamos o caso em que os cortes são do tipo não-guilhotinado. Também inserimos restrições como: permitir a rotações dos itens; satisfazer uma dada ordem de

descarregamento; satisfazer às condições de estabilidade estática para corpos rígidos; e, manter o centro de gravidade dentro de uma região considerada "segura".

Os problemas bidimensionais do parágrafo anterior foram investigados para itens/recipientes com a forma retangular. Entretanto, algumas variantes bidimensionais em que os itens assumem forma irregular também foram consideradas.

Já nas versões tridimensionais, o tipo de corte adotado foi o guilhotinado e os itens tinham forma regular. Lidamos com o caso em que os itens podem ser rotacionados em torno de todos os eixos e a versão com estágios de corte. Consideramos também o problema do corte de estoque com recipientes de tamanhos variados *Cutting Stock with Variable Bin Sizes Problem*.

1.2.3 Algumas Técnicas para Problemas em Otimização

Discutiremos brevemente algumas técnicas empregadas para solucionar problemas em otimização, sendo algumas delas utilizadas nesta tese. As técnicas citadas incluem programação dinâmica, o método *simplex*, o método *simplex* com geração de colunas, *branch-and-bound*, *branch-and-cut*, *branch-and-price* e *branch-and-cut-and-price*.

O custo dos algoritmos será medido através da complexidade de tempo e de espaço como funções do tamanho da entrada do respectivo problema. Usamos a notação O. Desta forma, um algoritmo terá complexidade de tempo polinomial se a função que descreve sua complexidade de tempo for limitada por um polinômio no tamanho da entrada. O mesmo se aplica à complexidade de espaço.

A técnica de programação dinâmica baseia-se na ideia de *divisão-e-conquista* e consiste em: dividir o problema original em subproblemas semelhantes ao original; calcular e armazenar em uma tabela a solução ótima de cada subproblema; e combinar a solução dos subproblemas para formar a solução ótima do problema original.

O método *simplex* é um procedimento matricial usado na resolução de *problemas de programação linear* (ou programas lineares). Um programa linear é aquele cuja função objetivo e restrições são todas lineares. Ele é chamado de *programa linear inteiro* quando as variáveis assumem apenas valores inteiros. O consumo de tempo deste método é exponencial no pior caso. Porém, é considerado bastante eficaz pela literatura especializada ([7, 25]).

A técnica de geração de colunas foi proposta inicialmente por [31]. Porém, foi no trabalho de Gilmore e Gomory (1961; 1963; 1965) [41, 42, 43] aplicado aos problemas corte de estoque uni- e bidimensionais que o método mostrou ser uma ferramenta poderosa para solucionar programas lineares. Ela é interessante para resolver problemas em programação linear quando o número de colunas da matriz relacionada ao programa é proibitivamente grande. Em vez de armazenar todas as colunas do sistema linear, esta técnica mantém um conjunto bem limitado de colunas, tipicamente a base da solução atual. Então, a cada iteração resolve um problema correspondente à busca da coluna a ser inserida pelo método Simplex no intuito de melhorar a

solução atual. É importante mencionar que a aplicação do método de geração de colunas pode resultar em soluções fracionárias, já que este é aplicado na relaxação linear do problema. Por isso, depois de resolvido o programa linear, um novo problema aparece e consiste em determinar uma solução inteira a partir da solução fracionária.

Os métodos *branch-and-bound* são geralmente empregados na busca de soluções exatas em otimização combinatória e tratam o processo de busca de uma solução semelhante a busca em uma árvore. Em linhas gerais, tais métodos resolvem relaxações do problema para obter limitantes superiores e inferiores das soluções. A partir disso, faz-se a "poda" de ramos da árvore sempre que soluções ingressem em um tal ramo fora dos limitantes obtidos. O conceito de *branch* ocorre quando, a partir da solução relaxada (ou parcial) atual, deve-se gerar novas soluções em que são testados novos valores com relação aos limitantes obtidos.

Outra método utilizado consiste no *branch-and-cut*. Este combina os métodos *branch-and-bound* com planos de corte. O algoritmo *branch-and-cut* é similar ao algoritmo *branch-and-bound*, porém faz a inclusão de inequações válidas em cada nó na tentativa de melhorar os limitantes. Com isso, visa reduzir o número de nós da árvore *branch-and-bound*.

Estudos recentes têm combinado a técnica de geração de colunas com o método *branch-and-bound*, criando estratégias conhecidas como *branch-and-price*. O método *branch-and-price* é uma variação do método *branch-and-bound* em que é usada programação linear junto com método de geração de colunas para gerar os limitantes. Quando um ótimo fracionário é encontrado, a operação de *branch* acontece adicionando novas restrições, as quais dividem o problema em subproblemas com mais restrições e com o intuito de forçar a integralidade. Fases de *branching* e de otimização (através do método de geração de colunas) são realizadas gerando novas soluções até alcançar o ótimo.

Outros estudos têm realizado a combinação do *branch-and-price* com planos de cortes, gerando os métodos *branch-and-cut-and-price*. Nele se considera, além da inserção de colunas, planos de cortes. Apesar deste método fornecer um problema linear com relaxações fortes, fazer a geração de colunas com planos de corte não é uma tarefa simples, já que a geração de colunas pode se tornar muito difícil depois da adição de restrições. Note que, ao adicionar novas restrições, podemos destruir a estrutura do problema associado à geração de colunas. Encontramos poucos trabalhos na literatura (como em [5, 13]) que empregam este método, dentro da área de problemas de corte e empacotamento.

1.3 Organização da Tese

Este tese é formada por uma coletânea de artigos. Eles foram escritos com o intuito de serem publicados em congressos e/ou revistas especializadas da área.

Os artigos estão apresentados nos próximos capítulos, sendo cada capítulo formado por um artigo. Resumidamente, cada artigo contém a descrição dos problemas tratados juntamente

com as restrições adotadas, uma breve revisão da literatura e, é claro, os resultados obtidos, podendo incluir demonstrações, algoritmos e/ou modelos de programação linear. Além disso, resultados experimentais considerando diversas instâncias são apresentados. Faremos um breve relato destes artigos nas subseções seguintes.

Excetua-se dos capítulos anteriores o último capítulo, Capítulo 7, que traz a conclusão e sugestões para trabalhos futuros.

1.3.1 Resultados do Capítulo 2

O próximo capítulo lida com a versão tridimensional dos seguintes problemas: mochila irrestrita; corte de estoque; corte de estoque com recipientes de tamanhos variados; e, empacotamento em faixa. Também lidamos com a versão destes problemas que permitem a rotação ortogonal dos itens em torno de todos os eixos e a versão em que os cortes devem ser feitos em estágios. Estes problemas foram tratados usando a terminologia de corte guilhotinado.

Seguimos a mesma linha proposta em Cintra *et al.* (2008) [27]. Para a versão do problema da mochila irrestrita e suas variantes, apresentamos algoritmos exatos baseado na técnica de programação dinâmica e que faz uso do conceito de *reduced raster points* [83]. Os algoritmos obtidos mostraram ser bastante eficientes, resolvendo a otimalidade diversas instâncias em pouco tempo computacional. Muitas delas foram resolvidas em tempo inferior a $0,01$ segundos e a ocupação do recipiente ficou acima dos 85% na média. No caso com rotações, esta ocupação ultrapassou os 94% na média, incluindo a versão com estágios de corte.

Para a versão do problema do corte de estoque e variantes, apresentamos uma heurística baseada na técnica de geração de colunas. Esta heurística faz uso dos algoritmos exatos do problema da mochila irrestrita para gerar as colunas. As soluções inteiras foram obtidas por meio da resolução da relaxação do programa inteiro, da reaplicação do método em instâncias residuais e por uma heurística primal construtiva. Uma extensão desta heurística foi feita para lidar com as versões dos problemas do corte de estoque com recipientes de tamanhos variados e empacotamento em faixa. Através dos experimentos computacionais comprovamos a eficiência desta heurística quando temos instâncias com itens de grande demanda. As soluções obtidas ficaram a menos 3% do ótimo, na média. No caso sem rotações, esta diferença não ultrapassou os 1% na média e o consumo de tempo foi razoavelmente baixo.

O artigo apresentado no Capítulo 2 foi submetido para uma revista da área. Uma versão preliminar foi apresentada e publicada nos anais do XLI SBPO - Simpósio Brasileiro de Pesquisa Operacional, realizado na Cidade de Porto Seguro - BA, em setembro de 2009. Este trabalho foi realizado em colaboração com Yoshiko Wakabayashi (IME-USP) e Eduardo C. Xavier (IC-UNICAMP) [80].

1.3.2 Resultados do Capítulo 3

No Capítulo 3 apresentamos resultados para os problemas da mochila irrestrita e do corte de estoque nas suas versões bidimensionais. Lidamos com o caso em que os cortes devem ser ortogonais e não-guilhotinados, ou seja, os cortes não são restritos a serem efetuados por uma guilhotina, mas devem ser ortogonais aos lados do recipiente que está sendo cortado. As versões destes problemas que permitem a rotação ortogonal dos itens também foi considerada.

No caso dessa versão do problema da mochila irrestrita e sua variante que permite a rotação ortogonal dos itens, apresentamos três estratégias para resolvê-las, a saber: a Heurística de Cinco-Blocos Recursiva; a Abordagem $L^{(k)}$; e, a Abordagem L. A ideia da Heurística de Cinco-Blocos Recursiva é dividir recursivamente um retângulo em cinco (ou menos) retângulos menores através de *cortes não-guilhotinados de primeira ordem*. A abordagem L realiza o corte/empacotamento dos itens em retângulos e em peças no formato de L. Em testes computacionais preliminares, nossa versão da abordagem L trouxe bons resultados, apesar do moderado gasto de tempo computacional (algumas vezes alto consumo de tempo). Então, decidimos limitar tal abordagem para conter no máximo k níveis de recursão. Então, criamos a Abordagem $L^{(k)}$.

Os algoritmos acima foram combinados com o conceito de *reduced raster points*, e para obter o limitante inferior da solução usamos o algoritmo exato, baseado em programação dinâmica, para as variantes bidimensionais guilhotinada, proposto em [27]. Provamos a questão em aberto descrita no trabalho de Birgin *et al.* (2010) [17], que diz que não há perda de generalidade em considerar cortes feitos somente sobre os *reduced raster points* na abordagem L. Também apresentamos um contra-exemplo para a abordagem L de modo a concluir que ela não é capaz de sempre gerar soluções não-guilhotinadas ótimas para o problema em questão.

Estes algoritmos foram codificados e vários testes computacionais realizados em instâncias adaptadas de outros trabalhos da literatura. Obtivemos resultados que se mostraram bastante satisfatórios: conseguimos melhorar o valor da solução já conhecido de quase todas as instâncias consideradas.

Elaboramos um artigo em conjunto com Yoshiko Wakabayashi (IME-USP) descrevendo os resultados obtidos acima, veja [79]. Este foi apresentado e publicado nos anais do CLEI 2010 - XXXVI Conferência Latino-americana de Informática, ocorrido na Cidade de Assunção - Paraguai, em outubro de 2010. Vale mencionar que este artigo foi selecionado como um dos três melhores artigos de toda a conferência.

Em relação a versão do problema do corte de estoque, adaptamos a heurística, baseada em geração de colunas, proposta por Cintra *et al.* (2008) [27]. Deste modo, a geração de colunas passa a ser feita em até três fases por algoritmos desenvolvidos para a versão do problema da mochila irrestrita em estudo, a saber: (i) inicialmente, pelo algoritmo exato proposto por [27] para a versão bidimensional guilhotinada; (ii) caso nenhuma coluna seja adequada, usamos a Heurística de Cinco-Blocos Recursiva; (iii) e, se ainda nenhuma coluna for adequada, tentamos

a Abordagem $L^{(k)}$. Com os bons resultados da abordagem L, decidimos também usá-la na rotina de geração de colunas para obter a solução e o limitante inferior da solução ótima.

Os testes computacionais realizados com a heurística de geração de colunas produziram resultados satisfatórios. Para o caso sem rotações ortogonais dos itens, a solução ótima foi encontrada para todas as instâncias (exceto para três delas) ao usar somente a Abordagem L na rotina de geração de colunas. Além do mais, o tempo computacional requerido ficou abaixo dos 130 segundos no pior caso. Para o caso com rotações ortogonais, a solução obtida diferiu do limitante inferior em apenas uma unidade e o tempo computacional, na média, ficou abaixo dos 280 segundos.

A versão completa do artigo que contempla ambos os problemas está descrita no Capítulo 3. Este artigo também foi escrito em colaboração com Yoshiko Wakabayashi (IME-USP).

1.3.3 Resultados do Capítulo 4

O terceiro resultado obtido nesta tese e apresentado em forma de artigo no Capítulo 4 contempla a variante bidimensional do problema de empacotamento em faixa em que não há restrição quanto ao tipo de corte e apenas a versão com orientação fixa é considerada. Neste caso, requeremos que os itens sejam empacotados de forma ortogonal.

Para o problema em questão, duas restrições adicionais foram consideradas: o empacotamento deve satisfazer uma ordem de descarregamento, isto é, a restrição de ordem; e o empacotamento deve ser estável, ou seja, atender as condições de equilíbrio estático de corpos rígidos.

Em relação ao estudo da estabilidade dentro de problemas de Corte e Empacotamento observando às condições reais de equilíbrio estático, encontramos na literatura apenas o trabalho de Castro Silva *et al.* (2003) [85].

Diferente do apresentado por Castro Silva *et al.* (2003) em relação à estabilidade, nossa metodologia permite, além de verificar a estabilidade estática do empacotamento, calcular o quanto de força (peso/carga) é passado de um item para os demais. Nesta perspectiva, nossa metodologia permite ser estendida para também lidar com os casos de estabilidade dinâmica, ou seja, nos casos em que o recipiente/itens tenham variações de velocidade, inclinações e/ou curvaturas.

Nosso objetivo para este problema consistiu em obter soluções exatas (empacotamentos ótimos) que atendessem às restrições impostas. Como a restrição de estabilidade apresentou formulações complexas, as quais inviabilizaram criar um de modelo de programação linear que as considerasse, optamos por um algoritmo *branch-and-cut*. A formulação inteira proposta fornece um empacotamento que atende à todas as restrições citadas acima, exceto às de estabilidade. Para evitar soluções inteiras que correspondem a empacotamentos instáveis foram inseridas restrições (planos de corte) para as quais estas soluções tornam-se inviáveis.

Além do algoritmo *branch-and-cut*, três heurísticas foram propostas: a primeira realiza o empacotamento dos itens em níveis verticais, enquanto a segunda é derivada do algoritmo *branch-and-bound* proposto por Martello *et al.* (2000) [68] e se baseia na ideia de empacotar itens em pontos denominados *pontos de canto*. A terceira heurística é obtida a partir do algoritmo *branch-and-cut* proposto.

Durante a realização dos experimentos computacionais, fizemos a combinação destas heurísticas com o algoritmo *branch-and-cut* com o intuito de acelerar este último algoritmo. Os experimentos computacionais reportados indicam a aplicabilidade do algoritmo *branch-and-cut* para resolver instâncias de pequeno a médio porte. Enquanto, algumas das heurísticas podem ser aplicadas para resolver instâncias de grande porte, visto terem resolvido a otimalidade várias instâncias requerendo pouco tempo computacional.

Um artigo com parte dos resultados acima foi apresentado e publicado nos anais do CLEI 2010 - XXXVI Conferência Latino-americana de Informática, ocorrido na Cidade de Assunção - Paraguai, em outubro de 2010. A versão do CLEI 2010 foi escrita em conjunto com Fabrício L. S. da Silva [84]. Por outro lado, o Capítulo 4 traz a versão completa deste artigo.

1.3.4 Resultados do Capítulo 5

Uma extensão da formulação apresentada no Capítulo 4 enquanto tratando daquela variante do problema de empacotamento em faixa é utilizada para lidar com outra variante bidimensional deste mesmo problema.

No Capítulo 5, então, discorremos sobre a variante bidimensional do problema de empacotamento em faixa com as restrições: os itens devem ser empacotados de forma ortogonal aos lados do recipiente; os itens não podem ser rotacionados; e a restrição de ordem deve ser atendida. Além destas restrições, foi considerada uma outra restrição que envolve o balanceamento do empacotamento.

A restrição sobre o balanceamento do empacotamento requer que o centro de gravidade do empacotamento seja mantido dentro de uma região considerada "segura". Porém, não queremos que apenas o empacotamento final tenha seu centro de gravidade dentro desta região, mas que subempacotamentos envolvendo certos conjuntos de itens também satisfaçam esta restrição. Esta restrição possuem fortes implicações no planejamento de cargas de aviões, barcos, caminhões, etc., já que implica em segurança, por exemplo, em manter a estabilidade durante o transporte, bem como em redução dos gastos, como redução no consumo de combustível [59, 72].

Adaptamos o modelo de programação inteira apresentado no Capítulo 4 para tratar desta variante do problema de empacotamento em faixa. Nesta versão do problema foi possível inserir as restrições relacionadas ao balanceamento de carga dentro do modelo de programação linear inteira. Os resultados computacionais mostraram que o modelo é adequado para instâncias

pequenas e médias.

Um resumo estendido do artigo descrito no Capítulo 5, o qual contém os resultados citados acima, foi elaborado e submetido para um simpósio da área.

1.3.5 Resultados do Capítulo 6

O último capítulo envolvendo os resultados obtidos contempla os seguintes problemas: mochila 0/1, mochila irrestrita e corte de estoque em suas versões bidimensionais.

Estes problemas foram tratados para o caso em que os itens são representados por polígonos convexos e não-convexos, ou seja, consideramos a versão com itens irregulares e, em todos os casos, o recipiente possui forma retangular.

Para lidar com a geometria nestes problemas foi utilizado o conceito de *No-Fit polygon*, sendo implementada a robusta estratégia proposta por Burke *et al.* (2007) [19]. Esta estratégia foi combinada com uma versão estendida do algoritmo de busca de Adamowicz e Albano (1976) [1] com o intuito de encontrar o melhor ponto para empacotar um item irregular dentro do recipiente. Nesta perspectiva, o objetivo era encontrar um ponto que minimizasse a área da envoltória retangular e convexa do empacotamento.

Em relação a variante do problema da mochila 0/1 considerada, propusemos uma heurística baseada em GRASP. A ideia foi gerar uma solução gulosa inicial por meio de amostras aleatórias de itens e, através da avaliação de soluções vizinhas, buscava-se melhorar a solução atual. Com esta heurística foi possível obter a solução ótima para 7 das 15 instâncias consideradas nos experimentos computacionais. O tempo médio computacional requerido pela heurística foi pequeno para grande parte das instâncias.

Para a versão do problema da mochila irrestrita, nossa estratégia dividiu-se em dois passos: primeiramente, subconjuntos de itens irregulares foram empacotados em retângulos pequenos com alta taxa de ocupação. De posse destes retângulos, criamos uma instância para versão bidimensional do problema da mochila irrestrita que lida com itens retangulares. Usamos o algoritmo exato proposto por Cintra *et al.* (2008) [27] para obter o empacotamento final. Nos resultados computacionais deste caso obtivemos soluções com taxa de ocupação do recipiente acima dos 90% para mais de 60% das instâncias consideradas.

Por fim, a variante do problema do corte de estoque para itens irregulares foi resolvida por uma versão adaptada da heurística de geração de colunas de [27]. As colunas foram geradas pelo algoritmo desenvolvido para a versão do problema da mochila irrestrita discutida no parágrafo anterior. Sempre que uma coluna precisava ser computada, um novo subconjunto de retângulos pequenos era obtido observando o valor corrente das variáveis duais. Soluções iguais ao limitante inferior da solução ótima foram obtidas para 6 das 15 instâncias usadas. Para os outros casos, a solução obtida diferiu de no máximo duas unidades do limitante inferior. Em contrapartida, o tempo requerido para resolver as instâncias foi alto. Na média, ficou próximo

dos 42.600 segundos.

Um artigo envolvendo estes resultados foi escrito e está apresentado no Capítulo 6. Este artigo teve a colaboração de Aline M. Del Valle e Eduardo C. Xavier (IC-Unicamp).

Chapter 2

Algorithms for 3D Guillotine Cutting Problems: Unconstrained Knapsack, Cutting Stock and Strip Packing

Abstract

We present algorithms for the following three-dimensional (3D) guillotine cutting problems: Unconstrained Knapsack, Cutting Stock and Strip Packing. We consider the case where the items have fixed orientation and the case where orthogonal rotations around all axes are allowed. For the Unconstrained 3D Knapsack problem, we extend the recurrence formula proposed by Beasley (1985) [8] for the Rectangular Knapsack Problem and present a dynamic programming algorithm that uses *reduced raster points*. We also consider a variant of the Unconstrained Knapsack problem in which the cuts must be staged. For the 3D Cutting Stock problem and its variants in which the bins have different sizes (and the cuts must be staged), we present column generation based algorithms. Modified versions of the algorithms for the 3D Cutting Stock problems with stages are then used to build algorithms for the 3D Strip Packing problem and its variants. The computational tests performed with the algorithms described in this paper indicate that they are useful to solve instances of moderate size.

2.1 Introduction

The problem of cutting large objects to produce smaller objects has been largely investigated, specially when the objects are one- or two-dimensional. We focus here on the three-dimensional

case, restricted to guillotine cuts. In this context, the large objects to be cut are called *bins*, and the small objects (to be produced) are called *boxes* or *items*.

A *guillotine cut* is a cut that is parallel to one of the sides of the bin and goes from one side to the opposite one. For the problems considered here, not only the first cut, but all the subsequent cuts on the smaller parts must be of guillotine type.

A *k-staged cutting* is a sequence of at most k stages of cuts, each stage of which is a set of parallel guillotine cuts performed on the objects obtained in the previous stage. Moreover, the cuts in each stage must be orthogonal to the cuts performed in the previous stage. We assume, without loss of generality, that the cuts are infinitely thin.

Each possible way of cutting a bin is called a *cutting pattern* (or simply, *pattern*). To represent the patterns (and the cuts to be performed), we consider the Euclidean space $\mathbb{R}^3$ with the xyz coordinate system, and assume that the length, width and height of an object is represented in the axes x, y and z, respectively. We say that a bin (or box) B has dimension (L, W, H), and write $B = (L, W, H)$, if it has length L, width W and height H. For such a bin, we assume that the position $(0, 0, 0)$ corresponds to its bottom-left front corner, and position (L, W, H) represents its top-right behind corner. Analogously, the same terminology is used for the boxes.

The problems considered in this paper are the following.

THREE-DIMENSIONAL UNCONSTRAINED KNAPSACK PROBLEM (3KPG): We are given a bin $B = (L, W, H)$ and a list T of n types of boxes, each type i with dimension (l_i, w_i, h_i) and value v_i, $i = 1, \ldots, n$. We wish to determine how to cut B to produce boxes of some of the types in T so as to maximize the total value of the boxes that are produced. Here, no bound is imposed on the number of boxes of each type that can be produced (some types may not occur). An instance of this problem is denoted by a tuple (L, W, H, l, w, h, v), where $l = (l_1, \ldots, l_n)$ and w, h and v are lists defined likewise.

In the problem 3KPG there is no demand associated with a box. Differently, in the cutting stock and strip packing problems, to be defined next, there is a demand associated with each type of box. In this case, a box of type i with dimension (l_i, w_i, h_i) and demand d_i is denoted by a tuple (l_i, w_i, h_i, d_i); and a set of n types of boxes is denoted by (l, w, h, d), where $l = (l_1, \ldots, l_n)$, and w, h and d are lists defined analogously.

THREE-DIMENSIONAL CUTTING STOCK PROBLEM (3CSG): Given an unlimited quantity of identical bins $B = (L, W, H)$ and a set of n types of boxes (l, w, h, d), determine how to cut the smallest possible number of bins B so as to produce d_i units of each box type i, $i = 1, \ldots, n$. An instance for this problem is given by a tuple (L, W, H, l, w, h, d).

THREE-DIMENSIONAL CUTTING STOCK PROBLEM WITH VARIABLE BIN SIZES (3CSVG): Given an unlimited quantity of b different types of bins $B_1, \ldots, B_b$, each bin B_j with dimension (L_j, W_j, H_j) and value V_j, and a set of n types of boxes (l, w, h, d), determine how to cut the

given bins to generate d_i units of each box type i, $i = 1, \ldots, n$, so that the total value of the bins used is the smallest possible. (Some types of bins may not be used.) An instance of this problem is given by a tuple (L, W, H, V, l, w, h, d).

THREE-DIMENSIONAL STRIP PACKING PROBLEM (3SPG): Given a 3D strip $B = (L, W, \infty)$ (a bin with bottom dimension (L, W) and infinite height) and a set of n types of boxes (l, w, h, d), determine how to cut the strip B so that d_i units of each box type i, $i = 1, \ldots, n$, is produced and the height of the part of the strip that is used is minimized. We require the cuts to be k-staged (and horizontal in the first stage); furthermore, the distance between any two subsequent cuts must be at most A (a common restriction imposed by the cutting machines).

For the problems above mentioned, we also consider variants in which orthogonal rotations of the boxes are allowed. These variants are called 3KPG^r, 3CSG^r, 3CSVG^r and 3SPG^r, respectively. When we allow a box $b_i = (l_i, w_i, h_i)$ to be rotated, this means that its dimension can be considered as being any of the six permutations of (l_i, w_i, h_i). We represent the versions of these problems that do not specify the type of cut by 3KP, 3CS, 3CSV and 3SP.

Throughout the paper, the dimensions of the bins and the boxes are assumed to be integer. For the staged variant of the 3CSG problem, we assume that the first cutting stage is performed in the horizontal direction, that is, parallel to the xy-plane, denoted as $'H'$; followed by a cut in the lateral vertical direction, that is, parallel to the yz-plane, denoted as $'V'$; and then, a cut in the frontal vertical direction (parallel to the xz-plane), denoted as $'D'$ (a depth cut).

All problems above mentioned are $\mathcal{NP}$-hard. The one- and two-dimensional versions of the unconstrained knapsack problem have been studied since the sixties. Herz (1972) [48] presented a recursive algorithm for the two-dimensional version, called 2KP, which obtains canonical patterns making use of *discretization points*. Beasley (1985) [8] proposed a dynamic programming formulation that uses the discretization points to solve the staged and non-staged variants of the 2KP problem. Cintra *et al.* (2008) [27] presented a dynamic programming approach for the 2KP problem and some of its variants. They were able to solve in a small computational time instances of the OR-Library for which no optimal solution was known. Diedrich *et al.* (2008) [32] proposed approximation algorithms for the 3KP problem with approximation ratios $(9+\epsilon)$, $(8 + \epsilon)$ and $(7 + \epsilon)$; and for the 3KP^r problem they designed an approximation algorithm with ratio $(5 + \epsilon)$.

The first column generation approaches for the one- and two-dimensional Cutting Stock problem, called 1CS and 2CS, were proposed by Gilmore and Gomory (1961; 1963; 1965) [41, 42, 43]. They also considered the variant of 2CS in which the bins have different sizes, called 2CSV, and proposed the k-staged version. Alvarez-Valdes *et al.* (2002) [4] also investigated the 2CS problem, for which they presented a column generation based algorithm that uses the recurrence formulas described in Beasley (1985) [8]. Puchinger and Raidl (2007) [78] presented a branch-and-price algorithm for the 3-staged case of 2CS.

For the 3CS problem with unit demand, Csirik and van Vliet (1993) [29] presented an algorithm with asymptotic performance ratio of at most 4.84. Miyazawa and Wakabayashi (2009) [71] showed that the version with orthogonal rotation is as difficult to approximate as the oriented version, and they also presented a 4.89-approximation algorithm for this case. Cintra *et al.* (2007) [26] showed that these approximation ratios are also preserved in the case of arbitrary demands.

Some approximation algorithms have been proposed for the two-dimensional Strip Packing (2SP) problem. Kenyon and Rémila (2000) [61] presented an AFPTAS for the oriented case and Jansen and van Stee (2005) [57] proposed a PTAS for the case in which rotations are allowed. Other approaches like branch-and-bound and integer linear programming models have also been proposed in [50, 66, 67]. Cintra *et al.* (2008) [27] presented a column generation based algorithm for the staged 2SP problem with and without rotations. For the three-dimensional case (3SP), Jansen and Solis-Oba (2006) [56] proposed an algorithm with asymptotic ratio of $2 + \epsilon$.

The results we present in this paper are basically extensions of the approaches obtained by Cintra *et al.* (2008) [27], combined with the use of *reduced raster points* (an idea introduced by Scheithauer (1997) [83]). Section 2.2 focus on the Unconstrained Knapsack problems 3KPG, 3KPG^r and its variants in which the cuts must be k-staged. For all these problems we present exact dynamic programming algorithms. For the Cutting Stock problems 3CSG, 3CSG^r, 3CSVG and 3CSVG^r, we present in Sections 2.3 and 2.4 column generation based algorithms that use as a routine the algorithm proposed for the Unconstrained Knapsack problem. In Section 2.5 we focus on the 3SPG problem and its variants (with rotations and/or k-staged cuts). The algorithms for all these problems use a column generation technique. The computational experiments with the algorithms described here are reported in Section 2.6.

2.2 The 3D Unconstrained Knapsack Problem

The algorithms we describe in this section are based on the use of the so-called *raster points*. These are a special sub-set of the *discretization points* (positions where guillotine cutting can be performed) and were first presented by Scheithauer (1997) [83].

Discretization points were used (for the two-dimensional case) by Herz (1972) [48] and also by Beasley (1985) [8] in a dynamic programming algorithm. More recently, Birgin *et al.* (2010) [17] used raster points to deal with the packing of identical rectangles in another rectangle, obtaining very good results.

Let (L, W, H, l, w, h, v) be an instance of the 3KPG problem. A *discretization point of length* (respectively, *of the width* and *of the height*) is a value $i \leq L$ (respectively, $j \leq W$ and $k \leq H$) obtained by an integer conic combination of $l = (l_1, \ldots, l_n)$ (respectively, $w = (w_1, \ldots, w_n)$ and $h = (h_1, \ldots, h_n)$). We denote by P, Q and R the set of all discretization points of length, width and height, respectively.

The set of *reduced raster points* $\tilde{P}$ (relative to P) is defined as $\tilde{P} = \{\langle L - r\rangle : r \in P\}$, where $\langle s\rangle = \max\{t \in P : t \leq s\}$. In the same way we define the sets $\tilde{Q}$ (relative to Q) and $\tilde{R}$ (relative to R). To simplify notation, we refer to these points as *r-points*. An important feature of the r-points is the fact that they are sufficient to generate all possible cutting patterns (that is, for every pattern there is an equivalent one in which the cuts are performed only on r-points). As the set of r-points is a subset of the discretization points, this may reduce the time for the search of an optimum pattern. To refer to these points we define, for any rational number $x_r \leq L$, $y_r \leq W$ and $z_r \leq H$, the following functions:

$$\begin{array}{l} p(x_r) = \max\{i|\ i \in \tilde{P},\ i \leq x_r\}; \\ q(y_r) = \max\{j|\ j \in \tilde{Q},\ j \leq y_r\}; \\ r(z_r) = \max\{k|\ k \in \tilde{R},\ k \leq z_r\}. \end{array} \tag{2.1}$$

The algorithm to compute the r-points of a given instance is denoted by RRP. First, it generates the discretization points using the algorithm DDP (*Discretization using Dynamic Programming*) presented in [27], and then, it selects those that are r-points, following the above definition.

The time complexity of the algorithm RRP is the same of the algorithm DDP, that is, $O(nD)$ where $D := \max\{L, W, H\}$. This algorithm is pseudo-polynomial; so when D is small, or the dimensions of the boxes are not so small compared to the dimension of the bin, then the algorithm has a good performance, as shown by the computational tests, presented in Section 2.6.

2.2.1 Algorithm for the 3KPG problem

Let $I = (L, W, H, l, w, h, v)$ be an instance of the 3KPG problem, and let $\tilde{P}$, $\tilde{Q}$ and $\tilde{R}$ be the set of r-points, as defined previously. Let $G(L, W, H)$ be the value of an optimum guillotine pattern for the instance I. The function G can be calculated by the recurrence formula (2.2). In this formula, $g(l^*, w^*, h^*)$ denotes the maximum value of a box that can be cut in a bin of dimension (l^*, w^*, h^*). This value is 0 if no box can be cut in such a bin.

$$G(l^*, w^*, h^*) = \max \left\{ \begin{array}{l} g(l^*, w^*, h^*); \\ \max\{G(l', w^*, h^*) + G(p(l^* - l'), w^*, h^*)|\ \ l' \in \tilde{P},\ \ l' \leq l^*/2\}; \\ \max\{G(l^*, w', h^*) + G(l^*, q(w^* - w'), h^*)|\ \ w' \in \tilde{Q},\ \ w' \leq w^*/2\}; \\ \max\{G(l^*, w^*, h') + G(l^*, w^*, r(h^* - h'))|\ \ h' \in \tilde{R},\ \ h' \leq h^*/2\}. \end{array} \right\} \tag{2.2}$$

We note that the recurrence above is an extension of the recurrence formula of Beasley (1985) [8]. It can be solved by the algorithm DP3KPG (Dynamic Programming for the Three-dimensional Unconstrained Knapsack), which we describe next.

Algorithm 2.1: DP3KPG.

Input : An instance $I = (L, W, H, l, w, h, v)$ of the 3KPG problem.
Output : An optimum solution for I.

2.1.1 $\tilde{P} \leftarrow \text{RRP}(L, l), \quad \tilde{Q} \leftarrow \text{RRP}(W, w), \quad \tilde{R} \leftarrow \text{RRP}(H, h)$
2.1.2 Let $\tilde{P} = (p_1 < p_2 < \ldots < p_m),\ \tilde{Q} = (q_1 < q_2 < \ldots < q_s),\ \tilde{R} = (r_1 < r_2 < \ldots < r_u)$
2.1.3 **for** $i \leftarrow 1$ **to** m **do**
2.1.4 **for** $j \leftarrow 1$ **to** s **do**
2.1.5 **for** $k \leftarrow 1$ **to** u **do**
2.1.6 $G[i,j,k] \leftarrow \max(\{v_d |\ 1 \le d \le n;\ l_d \le p_i,\ w_d \le q_j \text{ and } h_d \le r_k\} \cup \{0\})$
2.1.7 $item[i,j,k] \leftarrow \max(\{d |\ 1 \le d \le n;\ l_d \le p_i,\ w_d \le q_j,\ h_d \le r_k \text{ and } v_d = G[i,j,k]\} \cup \{0\})$
2.1.8 $guil[i,j,k] \leftarrow nil$
2.1.9 **for** $i \leftarrow 1$ **to** m **do**
2.1.10 **for** $j \leftarrow 1$ **to** s **do**
2.1.11 **for** $k \leftarrow 1$ **to** u **do**
2.1.12 $nn \leftarrow \max(d |\ 1 \le d \le i \text{ and } p_d \le \lfloor p_i/2 \rfloor)$
2.1.13 **for** $x \leftarrow 1$ **to** nn **do**
2.1.14 $t \leftarrow \max(d |\ 1 \le d \le m \text{ and } p_d \le p_i - p_x)$
2.1.15 **if** $G[i,j,k] < G[x,j,k] + G[t,j,k]$ **then**
2.1.16 $G[i,j,k] \leftarrow G[x,j,k] + G[t,j,k]$
2.1.17 $pos[i,j,k] \leftarrow p_x$
2.1.18 $guil[i,j,k] \leftarrow 'V'$ // Vertical cut, parallel to yz-plane
2.1.19 $nn \leftarrow \max(d |\ 1 \le d \le j \text{ and } q_d \le \lfloor q_j/2 \rfloor)$
2.1.20 **for** $y \leftarrow 1$ **to** nn **do**
2.1.21 $t \leftarrow \max(d |\ 1 \le d \le s \text{ and } q_d \le q_j - q_y)$
2.1.22 **if** $G[i,j,k] < G[i,y,k] + G[i,t,k]$ **then**
2.1.23 $G[i,j,k] \leftarrow G[i,y,k] + G[i,t,k]$
2.1.24 $pos[i,j,k] \leftarrow q_y$
2.1.25 $guil[i,j,k] \leftarrow 'D'$ // Depth cut (vertical, parallel to xy-plane)
2.1.26 $nn \leftarrow \max(d |\ 1 \le d \le k \text{ and } r_d \le \lfloor r_k/2 \rfloor)$
2.1.27 **for** $z \leftarrow 1$ **to** nn **do**
2.1.28 $t \leftarrow \max(d |\ 1 \le d \le u \text{ and } r_d \le r_k - r_z)$
2.1.29 **if** $G[i,j,k] < G[i,j,z] + G[i,j,t]$ **then**
2.1.30 $G[i,j,k] \leftarrow G[i,j,z] + G[i,j,t]$
2.1.31 $pos[i,j,k] \leftarrow r_z$
2.1.32 $guil[i,j,k] \leftarrow 'H'$ // Horizontal cut, parallel to xy-plane
2.1.33 **return** $G(m, s, u)$.

First, the algorithm DP3KPG calls the algorithm RRP to compute the sets $\tilde{P}$, $\tilde{Q}$ and $\tilde{R}$ (lines $2.1.1 - 2.1.2$). Then (in the lines $2.1.3 - 2.1.8$), the algorithm stores in $G[i,j,k]$ for each bin of dimension (p_i, q_j, r_k), with $p_i \in \tilde{P}$, $q_j \in \tilde{Q}$ and $r_k \in \tilde{R}$, the maximum value of a box that can be cut in such a bin. The variable $item[i,j,k]$ indicates the corresponding box type, and the variable $guil[i,j,k]$ indicates the direction of the guillotine cut if its value is not *nil*. The value

nil indicates that no cut has to be performed, and $pos[i, j, k]$ contains the position (point) at x, y or z-axis where the cut has to be made.

Next, (in the lines $2.1.9 - 2.1.32$) the algorithm iteratively finds the optimum solution for a bin of the current iteration by the best combination of solutions already known for smaller bins. In other words, for a bin of dimension (p_i, q_j, r_k), the optimum solution is obtained in the following way: for each possible r-point p_x where a vertical cut $'V'$ can be performed, the algorithm determines the best solution by comparing the best solution so far with one that can be obtained with a vertical cut $'V'$ (lines $2.1.12 - 2.1.18$); repeat the same process for a depth cut $'D'$ (lines $2.1.19 - 2.1.25$), and for a horizontal cut $'H'$ (lines $2.1.26 - 2.1.32$). Finally, (at line $2.1.33$) the algorithm returns the value of an optimum solution.

The algorithm avoids generating symmetric patterns by considering, in each direction, r-points up to half of the size of the respective bin (see lines $2.1.12$, $2.1.19$ and $2.1.26$). In fact, consider a bin of width ℓ and an orthogonal guillotine cut in the x-axis at position $t \in \tilde{P}$, for $t > \frac{\ell}{2}$. This cut divides the current bin into two smaller bins: one with length t and the other with length $\ell - t$. The patterns that can be obtained with these two smaller bins can also be obtained using a guillotine cut at position $t' = \ell - t$ on the original bin. If $t' \in \tilde{P}$, then such a cut is considered as $t' \leq \frac{\ell}{2}$; if $t' \notin \tilde{P}$ then the cut at position $\langle t' \rangle$ generates two bins in which we can obtain the same patterns considered for the cut made on t'.

The time complexity of the algorithm DP3KPG is directly affected by the time complexity of the algorithm RRP (line 2.1.1). Therefore, the time complexity of the algorithm DP3KPG is $O(nL + nW + nH + m^2su + ms^2u + msu^2)$ where m, s and u are the total number of r-points of $\tilde{P}$, $\tilde{Q}$ and $\tilde{R}$, respectively. On the other hand, the space complexity of the DP3KPG is $O(L + W + H + msu)$.

2.2.2 Algorithm for the k-staged 3KPG problem

We present now a dynamic programming algorithm to solve the k-staged 3KPG and 3KPG^r problems. We consider that in each stage a different cut direction is considered, following the cyclic order: $H - V - D - H - \ldots$ A cutting stage may possibly be empty (when no cut has to be performed), and in this case, after it, the next cutting stage is considered.

In the next recurrence formulas, $G(l^*, w^*, h^*, k, V)$, $G(l^*, w^*, h^*, k, H)$ and $G(l^*, w^*, h^*, k, D)$ denote the value of an optimum guillotine k-staged solution for a bin of dimension (l^*, w^*, h^*). The parameters V, H and D indicate the direction of the first cutting stage.

$$G(l^*, w^*, h^*, 0, V \text{ or } H \text{ or } D) := g(l^*, w^*, h^*);$$

$$G(l^*, w^*, h^*, k, V) := \max \left\{ \begin{array}{l} G(l^*, w^*, h^*, k-1, D); \\ \max\{G(l', w^*, h^*, k-1, D) + G(p(l^* - l'), w^*, h^*, k, V) \mid \ l' \in \tilde{P}, \ l' \leq l^*/2\} \end{array} \right\},$$

$$G(l^*, w^*, h^*, k, H) := \max \left\{ \begin{array}{l} G(l^*, w^*, h^*, k-1, V); \\ \max\{G(l^*, w', h^*, k-1, V) + G(l^*, q(w^* - w'), h^*, k, H) \mid \ w' \in \tilde{Q}, \ w' \leq w^*/2\} \end{array} \right\}, \quad (2.3)$$

$$G(l^*, w^*, h^*, k, D) := \max \left\{ \begin{array}{l} G(l^*, w^*, h^*, k-1, H); \\ \max\{G(l^*, w^*, h', k-1, H) + G(l^*, w^*, r(h^* - h'), k, D) \mid \ h' \in \tilde{R}, \ h' \leq h^*/2\} \end{array} \right\}.$$

The algorithm DPS3KPG (Dynamic Programming for the k-staged 3KPG) described next solves the recurrence formulas above. It is very similar to the former algorithm (for the non-staged case). It computes first the sets $\tilde{P}$, $\tilde{Q}$ and $\tilde{R}$ and stores in $G[0, i, j, l]$ the maximum value of a box that can be cut on a bin of dimension (p_i, q_j, r_l) (lines $2.2.1-2.2.8$). Then, the algorithm computes, for each stage b, the best solution for cuts done only in one direction, and it uses this information to compute the best solution for the next stage, and so on (guaranteeing that two subsequent stages have cuts in different directions). This is the basic difference between the algorithm DP3KPG and DPS3KPG. In some cases, the best solution for the stage $b-1$ is also the solution for the stage b, and no cut is needed in this case. In this case, the value *nil* is stored in the variable *guil* (line $2.2.15$).

The algorithm DPS3KPG stores in $G[k, i, j, l]$ the optimum k-staged solution for a bin with dimension (p_i, q_j, r_l). The variables $guil[k, i, j, l]$, $pos[k, i, j, l]$ and $item[k, i, j, l]$ indicate, respectively, the direction of the first guillotine cut, the position of this cut at x, y or z-axis, and the corresponding item if no cut has to be made in the bin.

The time complexity of the algorithm DPS3KPG is the same of the algorithm DP3KPG multiplied by the number of cutting stages k. This is also true for the space complexity. On the other hand, if k is limited by some constant, then DPS3KPG have the same complexity of the algorithm DP3KPG.

2.2.3 The 3KPG^r problem and its variant with k stages

The problem 3KPG^r is a variant of 3KPG that allows orthogonal rotations of the boxes (to be cut) around any of the axes. This means that each box of type i can be considered as having one of the six dimensions obtained by the permutations of l_i, w_i, h_i (as long as they are feasible). We refer to these feasible dimensions as $\text{PERM}(l_i, w_i, h_i)$.

Algorithm 2.2: DPS3KPG.

Input : An instance $I = (L, W, H, l, w, h, v, k)$ of the k-staged 3KPG problem.
Output : An optimum k-staged solution for I.

2.2.1 $\tilde{P} \leftarrow \text{RRP}(L, l_{1,\ldots,n}), \quad \tilde{Q} \leftarrow \text{RRP}(W, w_{1,\ldots,n}), \quad \tilde{R} \leftarrow \text{RRP}(H, h_{1,\ldots,n})$
2.2.2 Let $\tilde{P} = (p_1 < p_2 < \ldots < p_m), \ \tilde{Q} = (q_1 < q_2 < \ldots < q_s), \ \tilde{R} = (r_1 < r_2 < \ldots < r_u)$
2.2.3 **for** $i \leftarrow 1$ **to** m **do**
2.2.4 **for** $j \leftarrow 1$ **to** s **do**
2.2.5 **for** $l \leftarrow 1$ **to** u **do**
2.2.6 $G[0, i, j, l] \leftarrow \max(\{v_d | \, 1 \le d \le n;\ l_d \le p_i,\ w_d \le q_j \text{ and } h_d \le r_l\} \cup \{0\})$
2.2.7 $item[0, i, j, l] \leftarrow \max(\{d | \, 1 \le d \le n;\ l_d \le p_i,\ w_d \le q_j,\ h_d \le r_l \text{ and } v_d = G[0, i, j, l]\} \cup \{0\})$
2.2.8 $guil[0, i, j, l] \leftarrow nil$
2.2.9 **if** $(k \ mod \ 3) = 1$ **then** $previous \leftarrow 'V'$ **else if** $(k \ mod \ 3) = 2$ **then** $previous \leftarrow 'D'$ **else** $previous \leftarrow 'H'$
2.2.10 **for** $b \leftarrow 1$ **to** k **do**
2.2.11 **for** $i \leftarrow 1$ **to** m **do**
2.2.12 **for** $j \leftarrow 1$ **to** s **do**
2.2.13 **for** $l \leftarrow 1$ **to** u **do**
2.2.14 $G[b, i, j, l] \leftarrow G[b-1, i, j, l]$
2.2.15 $guil[b, i, j, l] \leftarrow nil$
2.2.16 **if** $previous = 'D'$ **then**
2.2.17 $nn \leftarrow \max(d | \, 1 \le d \le m \ \text{ and } \ p_d \le \lfloor p_i/2 \rfloor)$
2.2.18 **for** $x \leftarrow 1$ **to** nn **do**
2.2.19 $t \leftarrow \max(d | \, 1 \le d \le m \ \text{ and } \ p_d \le p_i - p_x)$
2.2.20 **if** $G[b, i, j, l] < G[b-1, x, j, l] + G[b, t, j, l]$ **then**
2.2.21 $G[b, i, j, l] \leftarrow G[b-1, x, j, l] + G[b, t, j, l]$
2.2.22 $pos[b, i, j, l] \leftarrow p_x$
2.2.23 $guil[b, i, j, l] \leftarrow 'V'$
2.2.24 $previous \leftarrow 'V'$
2.2.25 **else if** $previous = 'V'$ **then**
2.2.26 $nn \leftarrow \max(d | \, 1 \le d \le u \ \text{ and } \ r_d \le \lfloor r_l/2 \rfloor)$
2.2.27 **for** $z \leftarrow 1$ **to** nn **do**
2.2.28 $t \leftarrow \max(d | \, 1 \le d \le u \ \text{ and } \ r_d \le r_l - r_z)$
2.2.29 **if** $G[b, i, j, l] < G[b-1, i, j, z] + G[b, i, j, t]$ **then**
2.2.30 $G[b, i, j, l] \leftarrow G[b-1, i, j, z] + G[b, i, j, t]$
2.2.31 $pos[b, i, j, l] \leftarrow r_z$
2.2.32 $guil[b, i, j, l] \leftarrow 'H'$
2.2.33 $previous \leftarrow 'H'$
2.2.34 **else**
2.2.35 $nn \leftarrow \max(d | \, 1 \le d \le s \ \text{ and } \ q_d \le \lfloor q_j/2 \rfloor)$
2.2.36 **for** $y \leftarrow 1$ **to** nn **do**
2.2.37 $t \leftarrow \max(d | \, 1 \le d \le s \ \text{ and } \ q_d \le q_j - q_y)$
2.2.38 **if** $G[b, i, j, l] < G[b-1, i, y, l] + G[b, i, t, l]$ **then**
2.2.39 $G[b, i, j, l] \leftarrow G[b-1, i, y, l] + G[b, i, t, l]$
2.2.40 $pos[b, i, j, l] \leftarrow q_y$
2.2.41 $guil[b, i, j, l] \leftarrow 'D'$
2.2.42 $previous \leftarrow 'D'$
2.2.43 **return** $G(k, m, s, u)$.

The problem 3KPG^r can be solved with the algorithms for the problem 3KPG. For that, we only need a preprocessing phase to change the instance. Given an instance I for the 3KPG^r, we construct another instance I' by adding to I, for each box i in I of dimension (l_i, w_i, h_i), the set of new types of boxes $\mathrm{PERM}(l_i, w_i, h_i)$, all with the same value v_i. Then, we solve the new instance I' with the algorithm 3KPG.

For the k-staged 3KPG^r problem, we proceed analogously. We denote the corresponding algorithms for these problems by DP3KPG^r and DPS3KPG^r.

2.3 The Three-dimensional Cutting Stock Problem

We first present some heuristics which will be used as subroutines in the column generation approach described in this section for the 3CSG problem. We also compare the sole performance of these heuristics with the performance of the column generation approach.

2.3.1 Primal heuristics for the three-dimensional cutting stock problem

The primal heuristic we present here – HFF3 – is a hybrid heuristic that generates patterns composed of levels. It uses an algorithm for the 2CS problem to generate the levels and an algorithm for the 1CS problem to pack these levels into bins. We first describe the algorithms for the 1CS and 2CS problems, and then we present the algorithm HFF3.

The algorithms for the 1CS problem that we use here are the well-known *First Fit* (FF), and *First Fit Decreasing* (FFD) algorithms. We describe here only the algorithm we use for the 2CS problem. It is called HFF2 (*Hybrid First Fit 2*), as it is based on the *Hybrid First Fit* algorithm, designed by Chung *et al.* (1982) [24]. (For convenience, we describe it as packing algorithm.)

The algorithm HFF2 includes two variants: HFF^l and HFF^w. Without loss of generality, we suppose that each box has unit demand. Thus, for an instance (L, W, l, w) of the 2CS problem, the algorithm HFF^l considers the items sorted decreasingly by length $(l_1 \geq l_2 \geq \ldots \geq l_n)$. Then, it considers each item i as a one-dimensional item of size w_i, and applies the algorithm *First Fit*, FF(W, w), to obtain a packing of those items into recipients $S_1, \ldots, S_m$, which we call *strips*. Finally, each strip S_i is considered as a one-dimensional item of size $s_i = \max\{l_j : j \in S_i\}$ and the algorithm FFD(L, s) is applied to pack these strips into rectangular (2D) bins. The strips of the algorithm HFF^l are generated in the length direction, whereas the HFF^w generates the strips in the width direction. The algorithm HFF2 executes both variants and returns a solution with the best value.

To deal with the 3CSG^r problem (the variant of 3CSG in which orthogonal rotations are allowed), we denote by HFF^x (respectively, HFF^y) the variant of the algorithm HFF2 that rotates the rectangles i to obtain $w_i \geq l_i$ (respectively, $l_i \geq w_i$) before applying the algorithms HFF^l and HFF^w. The algorithm HFF2^r executes these algorithms and returns the best solution found.

Instead of presenting the algorithm HFF3 directly, we present an algorithm called H3CS (see Algorithm 2.3) that uses as subroutines algorithms for the 1CS and 2CS problems. The algorithm HFF3 is a specialization of the algorithm H3CS using particular subroutines. The algorithm H3CS first sorts the items decreasingly by height. Then, it iteratively generates a new level using an algorithm for the 2CS problem, privileging the packing of the higher items into each level. For each item i, the largest possible number of them is packed without violating its demand and keeping the packing in one level (see line 2.3.7). When all levels are generated, they are packed into bins by an algorithm for the 1CS problem (see line 2.3.10).

We denote by HFF3$_h$ (respectively, HFF3$_h^r$) the algorithm H3CS that uses the algorithms FFD and HFF2 (respectively, HFF2^r) as subroutines. Observe that the algorithms HFF3$_h$ and HFF3$_h^r$ generate and pack the levels in the height direction. We denote by HFF3$_w$ and HFF3$_w^r$ (respectively, HFF3$_l$ and HFF3$_l^r$) the variants where levels are generated and packed in the width (respectively, length) direction. Finally, the algorithm HFF3 (respectively, HFF3^r) executes the algorithms HFF3$_h$, HFF3$_w$ and HFF3$_l$ (respectively, HFF3$_h^r$, HFF3$_w^r$ and HFF3$_l^r$) and returns the best packing obtained.

Algorithm 2.3: H3CS.

Input : An instance $I = (L, W, H, l, w, h, d)$ of the 3CSG problem.
Output : A solution for I.
Subroutines: Algorithms $\mathcal{A}$ and $\mathcal{B}$ for the 1CS and 2CS problems.

2.3.1 Sort the items of I decreasingly by height: $h_1 \geq h_2 \geq \ldots \geq h_n$.
2.3.2 $m \leftarrow 0$
2.3.3 **while** *exists* $d_i > 0$ *for some* $i \in \{1, \ldots, n\}$ **do**
2.3.4 $\quad m \leftarrow m + 1$
2.3.5 $\quad$ Let $d' = (d'_1, \ldots, d'_n)$ where $d'_i = 0$ for $i = 1, \ldots, n$
2.3.6 $\quad$ **for** $i \leftarrow 1$ **to** n **do**
2.3.7 $\quad\quad d'_i \leftarrow \max\{t : \ t \leq d_i,\ \widehat{d} = (d'_1, \ldots, d'_{i-1}, t, 0, \ldots, 0)$ and $|\mathcal{B}(L, W, l, w, \widehat{d})| \leq 1\}$
2.3.8 $\quad\quad d_i \leftarrow d_i - d'_i$
2.3.9 $\quad$ Let $N_m \leftarrow \mathcal{B}(L, W, l, w, d')$ and $h(N_m) = \max\{h_i : \ d'_i > 0\}$
2.3.10 Let $\mathcal{P}$ be a packing of the levels (N_i) in bins of height H by the algorithm $\mathcal{A}(H, h)$.
2.3.11 **return** $\mathcal{P}$

2.3.2 The column generation based heuristics

A well-known ILP formulation for the cutting stock problem uses one variable for each possible pattern [41]. This formulation is the following. Let $\mathcal{P}$ denote the set of cutting patterns and $m := |\mathcal{P}|$ denote its size. Now let P be an $n \times m$ matrix whose columns represent the cutting patterns, and P_{ij} indicates the number of copies of item i in pattern j. For each $j \in \mathcal{P}$, let x_j be the variable that indicates the number of times pattern j is used, and let d be the n-vector of

demands.

The following linear program is a relaxation of an ILP formulation for the cutting stock problem:

$$\begin{array}{l} \min \sum_{j\in\mathcal{P}} x_j \\ \text{subject to } \left\{ \begin{array}{ll} Px \geq d & \\ x_j \geq 0 & \text{for all } j \in \mathcal{P}. \end{array} \right. \end{array} \tag{2.4}$$

As we mentioned before, the column generation approach to solve the 1CS and 2CS problems was proposed in the early sixties by Gilmore and Gomory. The idea of this approach is to apply the Simplex method starting with a small set of columns of $\mathcal{P}$ as a basis, and generate new ones as needed. That is, in each iteration it obtains a new pattern (column) z with $\sum_{i=1}^{n} v_i z_i > 1$ such that z_i is the number of times box i appears in this pattern and v_i is the value of this box. After solving (2.4), one considers the integer part of the solution; and deal the residual problems iteratively using the same approach.

In the case of 3CSG we use the algorithm presented for the Three-dimensional Unconstrained Knapsack (3KPG) problem to generate such a pattern. In what follows, we describe the algorithm, denoted by Simplex$_{CS}$, that solves the linear program (2.4). In step 2.4.1, the matrix $I_{n\times n}$ is the identity matrix corresponding to n patterns, each one with items of one type and one orientation. More details about the column generation approach can be found in [25].

Algorithm 2.4: Simplex$_{CS}$.

Input : An instance $I = (L, W, H, l, w, h, d)$ of the 3CSG problem.
Output : An optimum solution for the linear program (2.4)
Subroutine: An algorithm $\mathcal{A}$ for the 3KPG or 3KPG^r problem.

2.4.1 Let $x \leftarrow d$ and $B \leftarrow I_{n\times n}$
2.4.2 Solve $y^T B = [1, 1, \ldots, 1]_n^T$
2.4.3 $z \leftarrow \mathcal{A}(L, W, H, l, w, h, y)$
2.4.4 **if** $y^T z \leq 1$ **then return** (B, x) **else** solve $Bw = z$
2.4.5 Let $t \leftarrow \min(\frac{x_j}{w_j} \mid 1 \leq j \leq n,\ \ w_j > 0)$ and $s \leftarrow \min(j \mid 1 \leq j \leq n,\ \ \frac{x_j}{wj} = t)$
2.4.6 **for** $i \leftarrow 1$ **to** n **do**
2.4.7 $\quad B_{i,s} \leftarrow z_i$
2.4.8 $\quad$ **if** $i = s$ **then** $x_i \leftarrow t$ **else** $x_i \leftarrow x_i - w_i\, t$
2.4.9 Go to line 2.4.2

We present below the algorithm CG3CSG that solves the 3CSG problem. It receives the solution (possibly fractional) found by the algorithm Simplex$_{CS}$ and returns an integer solution for the 3CSG problem. If needed, this algorithm uses a primal heuristic to obtain a cutting pattern that causes a perturbation of some residual instance (see line 14 in 2.5).

The algorithm CG3CSG solves (in each iteration) a linear system for an instance I and obtain B and x (line 2.5.1). Then, it obtains an integer vector x^*, just by rounding down the

Algorithm 2.5: CG3CSG.

Input : An instance $I = (L, W, H, l, w, h, d)$ of the 3CSG problem.
Output : A solution for I.
Subroutine: An algorithm $\mathcal{A}$ for the 3CSG problem or for the 3CSG^r problem.

2.5.1 $(B, x) \leftarrow \text{Simplex}_{CS}(L, W, H, l, w, h, d)$
2.5.2 **for** $i \leftarrow 1$ **to** n **do** $x_i^* \leftarrow \lfloor x_i \rfloor$
2.5.3 **if** *there is* i *such that* $x_i^* > 0$ *for some* $1 \leq i \leq n$ **then**
2.5.4 **return** $(B, x^*_{1,\ldots,n})$ (but do not halt)
2.5.5 **for** $i \leftarrow 1$ **to** n **do**
2.5.6 **for** $j \leftarrow 1$ **to** n **do** $d_i \leftarrow d_i - B_{i,j}x_j^*$
2.5.7 $n' \leftarrow 0,\ l' \leftarrow (\,),\ w' \leftarrow (\,),\ h' \leftarrow (\,),\ d' \leftarrow (\,)$
2.5.8 **for** $i \leftarrow 1$ **to** n **do**
2.5.9 **if** $d_i > 0$ **then**
2.5.10 $n' \leftarrow n' + 1,\ l' \leftarrow l' \| (l_i),\ w' \leftarrow w' \| (w_i),\ h' \leftarrow h' \| (h_i),\ d' \leftarrow d' \| (d_i)$
2.5.11 **if** $n' = 0$ **then HALT**
2.5.12 $n \leftarrow n',\ l \leftarrow l',\ w \leftarrow w',\ h \leftarrow h',\ d \leftarrow d'$
2.5.13 Go to line 2.5.1
2.5.14 **return** *a pattern of* $\mathcal{A}(L, W, H, l, w, h, d)$ *that has the largest volume, and update the demands (but do not halt).*
2.5.15 **if** *there exists* i $(1 \leq i \leq n)$ *such that* $d_i > 0$ **then** go to line 2.5.1

vector x (see line $2.5.2$). The vector x^* is a $'$partial$'$ solution that possibly fulfills only part of the demands. Thus, if there is a box i with part of its demand fulfilled by x^*, the algorithm returns (B, x^*), and the patterns corresponding to B. After this, the algorithm defines a new *residual instance* $I' = (L, W, H, l, w, h, d')$, where the vector $d' = (d'_1, \ldots, d'_n)$ contains the *residual demand* of each item i (see lines $2.5.7 - 2.5.12$). If d' is a null vector, then the algorithm halts (see line $2.5.11$), as this means that each item i has its demand fulfilled; otherwise, the execution proceeds to solve the new updated instance I.

The vector x returned by the algorithm Simplex_{CS} might have all components smaller than 1. In this case, x^* is a null vector and the subroutine $\mathcal{A}$ is used to obtain a good cutting pattern (line $2.5.14$). Therefore, the demands are updated and if there is some residual demand (lines $2.5.14 - 2.5.15$) the execution is restarted for the new residual instance (see line $2.5.1$). Note that the number of residual instances solved by the algorithm CG3CSG can be exponential in n. But, clearly, the algorithm halts because in each iteration the demands decrease. The algorithm $\mathcal{A}$ used as subroutine by the algorithm CG3CSG is the hybrid algorithm HFF3 described in Section 2.3.1.

An algorithm for the k-staged version of 3CSG can be obtained analogoulsy, just by changing the subroutine by the corresponding k-staged versions.

2.3.3 The 3CSG^r problem

We can solve the 3CSG^r problem also using the algorithms Simplex$_{CS}$ and CG3CSG, each one with the appropriate subroutines. Namely, in the Simplex$_{CS}$ we use the algorithm HFF3^r, and in the algorithm CG3CSG, we use the algorithm DP3KPG^r. We denote this version by CG3CSG^r. The same idea applies to the k-staged 3CSG^r problem in which we use the algorithm DPS3KPG as subroutine for the algorithm Simplex$_{CS}$.

2.4 The 3CSVG Problem

We can solve the 3CSVG problem using a column generation approach similar to the one described for the 3CSG problem. For that, basically we have to adapt the algorithm Simplex$_{CS}$.

In this problem we are given a list of different bins $B_1, \ldots, B_b$, each bin B_i with dimension (L_i, W_i, H_i) and value V_i, and we want to minimize the total value of the bins used to fulfill the demands. Using an analogous notation as before, the following is a relaxation of the integer linear program for the 3CSVG problem:

$$\begin{array}{l} \min \sum_{j\in\mathcal{P}} C_j x_j \\ \text{subject to } \left\{ \begin{array}{ll} Px \geq d & \\ x_j \geq 0 & \text{for all } j \in \mathcal{P}. \end{array} \right. \end{array} \tag{2.5}$$

The coefficient C_j in the above formulation indicates the value of the bin type used in pattern j. So, each C_j corresponds to some V_i.

Similarly to the 3CSG problem, if each box i has value y_i and occurs z_i times in a pattern j, we take a new column with $\sum_{i=1}^{n} y_i z_i > C_j$. Here, we can also use the algorithms we proposed for the Three-dimensional Unconstrained Knapsack problem to generate the (new) columns. The algorithm to solve (2.5) is called Simplex$_{CSV}$. The basic difference between the algorithms Simplex$_{CS}$ and Simplex$_{CSV}$ is that the latter has a vector f that associates one bin with each column of the matrix B. This vector and the variables B, *guil* and *pos* are used to reconstruct the solution found.

We describe now the algorithm CG3CSVG that solves the 3CSVG problem. It uses the algorithm Simplex$_{CSV}$ and is very similar to algorithm CG3CSG described for the 3CSG problem (we omit the details). The algorithm $\mathcal{A}$ used as subroutine by CG3CSVG is the hybrid algorithm HFF3.

The algorithm for the k-staged 3CSVG problem also uses the algorithm Simplex$_{CSV}$, but in this case with the subroutine for the k-staged 3KPG problem.

Algorithm 2.6: Simplex$_{CSV}$.

Input : An instance $I = (L, W, H, V, l, w, h, d)$ of the 3CSVG problem.
Output : An optimum solution for (2.5), where the columns of P are cutting patterns.
Subroutine: An algorithm $\mathcal{A}$ for the 3KPG or 3KPG^r problem.

2.6.1 Let f be a vector, where f_i is the smallest index j such that $l_i \leq L_j$, $w_i \leq W_j$ and $h_i \leq H_j$
2.6.2 Let $x \leftarrow d$ and $B \leftarrow I_{n\times n}$
2.6.3 $Solve\ y^T B = C_B^T$ // C_B is the vector $C = (C_1, \ldots, C_n)$ restricted to the columns of B
2.6.4 **for** $i \leftarrow 1$ **to** b **do**
2.6.5 $\quad z \leftarrow \mathcal{A}(L_i, W_i, H_i, l, w, h, y)$
2.6.6 $\quad$ **if** $y^T z > V_i$ **then** go to line 2.6.8
2.6.7 **return** $(B, f, x^*_{1,\ldots,n})$
2.6.8 $Solve\ Bw = z$
2.6.9 Let $t \leftarrow \min(\frac{x_j}{w_j} \mid 1 \leq j \leq n,\ \ w_j > 0)$ and $s \leftarrow \min(j \mid 1 \leq j \leq n,\ \ \frac{x_j}{wj} = t)$
2.6.10 Let $f_j = i$
2.6.11 **for** $i \leftarrow 1$ **to** n **do**
2.6.12 $\quad B_{i,s} \leftarrow z_i$
2.6.13 $\quad$ **if** $i = s$ **then** $x_i \leftarrow t$ **else** $x_i \leftarrow x_i - w_i\, t$
2.6.14 Go to line 2.6.3

Algorithm 2.7: CG3CSVG.

Input : An instance I = (L, W, H, V, l, w, h, d) of the 3CSVG problem.
Output : A solution for I.
Subroutine: An algorithm $\mathcal{A}$ for the 3CSVG problem or for the 3CSVG^r problem.

2.7.1 $(B, f, x) \leftarrow$ Simplex$_{CSV}(L, W, H, V, l, w, h, d)$
2.7.2 **for** $i \leftarrow 1$ **to** n **do** $x^*_i \leftarrow \lfloor x_i \rfloor$
2.7.3 **if** *there is i such that $x^*_i > 0$ for some $1 \leq i \leq n$* **then**
2.7.4 $\quad$ **return** $(B, f, x^*_{1,\ldots,n})$ (but do not halt)
2.7.5 $\quad$ **for** $i \leftarrow 1$ **to** n **do**
2.7.6 $\quad\quad$ **for** $j \leftarrow 1$ **to** n **do** $d_i \leftarrow d_i - B_{i,j} x^*_j$
2.7.7 $\quad n' \leftarrow 0,\ l' \leftarrow (\,),\ w' \leftarrow (\,),\ h' \leftarrow (\,),\ d' \leftarrow (\,)$
2.7.8 $\quad$ **for** $i \leftarrow 1$ **to** n **do**
2.7.9 $\quad\quad$ **if** $d_i > 0$ **then**
2.7.10 $\quad\quad\quad n' \leftarrow n' + 1,\ l' \leftarrow l' \| (l_i),\ w' \leftarrow w' \| (w_i),\ h' \leftarrow h' \| (h_i),\ d' \leftarrow d' \| (d_i)$
2.7.11 $\quad$ **if** $n' = 0$ **then HALT**
2.7.12 $\quad n \leftarrow n',\ l \leftarrow l',\ w \leftarrow w',\ h \leftarrow h',\ d \leftarrow d'$
2.7.13 $\quad$ Go to line 2.7.1
2.7.14 Let $V^* \leftarrow \min(\frac{V_i}{L_i W_i H_i}\ \mid i = 1, \ldots, f)$ and $j \leftarrow \min(i \mid \frac{V_i}{L_i W_i H_i} = V^*)$
2.7.15 **return** *a pattern of $\mathcal{A}(L_j, W_j, H_j, l, w, h, d)$ that has the largest volume, and update the demands.*
2.7.16 **if** *there exists i $(1 \leq i \leq n)$ such that $d_i > 0$* **then** go to line 2.7.1

2.4.1 The 3CSVG^r problem

For this problem, we use the algorithm CG3CSVG with the subroutine HFF3^r; and the algorithm Simplex$_{CSV}$ with the subroutine DP3KPG^r. This version of the algorithm is called

CG3CSVG^r. For the k-staged 3CSVG^r, problem we use the algorithm Simplex$_{CSV}$ with the subroutine DPS3KPG.

2.5 The Three-dimensional Strip Packing Problem

The 3D strip packing problem (3SPG) has been less tackled with the column generation approach. One advantage of this approach is that it is less sensitive to large values of demands. In the 3SPG problem the cuts must be k-staged, the first cutting stage has to be horizontal (that is, orthogonal to the height), and the distance between two subsequent cuts must be at most some given value A. We call A-*pattern* a guillotine cutting pattern between two subsequent horizontal cuts.

Let $\mathcal{P}$ be the set of all A-patterns, $|\mathcal{P}| = m$, and let A_j be the height of an A-pattern $j \in \mathcal{P}$.

The following is a relaxation of the integer linear program for the 3SPG problem:

$$\begin{array}{l} \min \sum_{j\in\mathcal{P}} A_j x_j \\ \text{subject to } \left\{ \begin{array}{ll} Px \geq d & \\ x_j \geq 0 & \text{for all } j \in \mathcal{P}. \end{array} \right. \end{array} \tag{2.6}$$

We can use the same approach presented for the 3CSVG problem to solve the 3SPG problem. For that, note that, each A-pattern of height A_j corresponds to a bin with dimension (L, W, A_j) and value precisely A_j in the 3CSVG problem. Thus, if $R = \{a_1, \ldots, a_b\}$ is the set of discretization points of height at most A, we can assume that $A = \max(a_1, \ldots, a_b)$, and we can consider that we are given b different types of bins (A-patterns), each one with dimension (L, W, a_j).

The algorithm to solve the k-staged 3SPG problem, called CG3SPG, is basically the algorithm presented for the k-staged 3CSVG problem with two modifications. First, to perturb the residual instance we generate a level with maximal volume (considering the height of such level). To do this, we use the algorithm HFF2 (for the 2CS problem). Second, every call to the algorithm Simplex$_{CSV}$ only solves one instance of the k-staged 3KPG problem, the one with dimensions (L, W, a_b). Observe that the variables G, $guil$ and pos computed by the algorithm DPS3KPG have the solutions for each height $a_i \in R$. This is an important modification because $|R|$ can be very large, and solving instances for each $a_i \in R$ considering a different bin would consume a lot of time.

For the k-staged 3SPG^r problem, we consider the algorithm HFF2^r to generate a perturbed instance. We also consider a modification in the algorithm HFF3 when we compare its solutions with the solution computed by the column generation algorithm. This modification basically consists in packing the levels generated by the algorithm HFF2 (or HFF2^r) one on top of the other in the direction z. We call M-HFF3 this modified algorithm. Finally, the maximum distance between two subsequent cuts is considered as the width of the bin.

2.6 Computational Tests

The tests were performed on several instances adapted from the literature. We present computational results for the set of instances adapted from Cintra *et al.* (2008) [27]. These instances were obtained in the following way: we considered the instances for the two-dimensional version of the problem, then we added the third dimension for each box (bin) by randomly choosing it from the dimensions already used for the other boxes (bins). These instances are available at the following url: `http://www.loco.ic.unicamp.br/binpack3d/`.

We only considered the first 12 instances, which we called $gcut1\_3d, \ldots, gcut12\_3d$. For each one the number of items and the dimensions of the bin are shown in Table 2.1. The length and width of the items were originally generated (see [8]) by sampling an integer from the uniform distribution $[25\% - 75\%]$ of the respective dimension (length and width) of the bin.

We also considered the set of 700 instances from Bischoff and Ratcliff (1995) [18]. In such work these instances were used in the Container Loading problem with the objective of maximizing the occupied volume of the container (we ignored the restriction that there were a limited number of copies of each item and the orientation restrictions). We used these instances only for the Knapsack problem, as we would not be able to show the results for each of the problems considered here. These instances were organized in groups of 100 instances. In each group, the dimensions of the container and the number of items are the same: only the dimensions of the items are different. For example, in the first group named thpack1, each instance consists of exactly 3 boxes, and the subsequent groups, thpack$2, \ldots,$ thpack7, have $5, \ldots, 20$ boxes, respectively. A standard ISO container of dimensions (587, 233, 220) is considered for all the instances. The average number of items per items type is 50.2 for the group thpack1, but decreases continuously and is only 6.5 for the group thpack7. The length, width, and height of the items are integers in the range of $[30 - 120]$, $[25 - 100]$ and $[20 - 80]$, respectively.

The algorithms presented in this paper were implemented in C language, and the tests were run on a computer with processor Intel® Core^{TM} 2 Quad 2.4 GHz, 4 GB of memory and operating system *Linux*. The linear systems in the column generation algorithms were solved by the Coin-OR CLP solver [28].

We are not aware of other works in the literature to compare our results. We did not find instances for the 3D Unconstrained Knapsack Problem, so we generated some to test our algorithms. We hope these instances will be useful to future researches on the 3D Unconstrained Knapsack Problem (and other problems) to perform comparative studies.

2.6.1 Comparing the use of raster points and discretization points

In this section we show, for some of the instances considered, the number of raster points, the number of discretization points, and the corresponding number of subproblems obtained. These numbers are shown in Table 2.1. We recall that m, s and u denotes the total number of

r-points (or discretization points) of length, width and height, respectively. The product msu gives the number of subproblems (#Subprob in the tables), where in the table we show the approximate number of subproblems where K stands for 10^3 and M for 10^6. In many cases it is very impressive the reduction on the number of subproblems that occurs with the use of the r-points. This has a great impact in the dynamic programming approach.

For the instances $gcut1\_3d$–$gcut12\_3d$, the number of subproblems using r-points were, on average, 0.77% of the number of subproblems using discretization points. For instances thpack1–thpack7, the columns with the numbers of r-points and discretization points indicate the average number (truncated) in each group. For the thpack instances, the number of subproblems using r-points corresponds, on average, to 19.01% of the number of subproblems using discretization points.

Instance	Number of items	Bin Dimensions	Raster Points m	s	u	#Subprob	Discretization Points m	s	u	#Subprob	#Subprob(%) Rast./Discr.
gcut1_3d	10	(250, 250, 250)	13	5	5	325	68	20	20	27.2K	1.19
gcut2_3d	20	(250, 250, 250)	17	24	13	5.3K	95	112	69	73.4K	0.72
gcut3_3d	30	(250, 250, 250)	44	26	22	25.1K	143	107	122	1.8M	1.35
gcut4_3d	50	(250, 250, 250)	45	50	29	65.2K	146	146	133	2.8M	2.30
gcut5_3d	10	(500, 500, 500)	10	13	8	1K	40	76	26	79K	1.32
gcut6_3d	20	(500, 500, 500)	12	18	8	1.7K	96	120	41	472.3K	0.37
gcut7_3d	30	(500, 500, 500)	23	19	17	7.4K	179	126	140	3.1M	0.24
gcut8_3d	50	(500, 500, 500)	44	59	27	70K	225	262	164	9.6M	0.73
gcut9_3d	10	(1000, 1000, 1000)	15	7	7	735	92	42	32	123.6K	0.59
gcut10_3d	20	(1000, 1000, 1000)	14	20	5	1.4K	89	155	37	510.4K	0.27
gcut11_3d	30	(1000, 1000, 1000)	20	38	14	10.6K	238	326	127	9.8M	0.11
gcut12_3d	50	(1000, 1000, 1000)	49	42	27	55.5K	398	363	291	42M	0.13
										AVERAGE	**0.77%**
thpack1	3	(587, 233, 220)	36	10	22	7.9K	100	27	53	143.1K	5.53
thpack2	5	(587, 233, 220)	88	65	48	274.5K	267	65	113	1.9M	14.00
thpack3	8	(587, 233, 220)	206	37	93	708.8K	390	114	155	6.8M	10.29
thpack4	10	(587, 233, 220)	263	52	110	1.5M	425	134	165	9.3M	16.01
thpack5	12	(587, 233, 220)	302	65	123	2.4M	445	146	172	11.1M	21.61
thpack6	15	(587, 233, 220)	339	81	134	3.6M	463	157	177	12.8M	28.60
thpack7	20	(587, 233, 220)	375	101	147	5.5M	481	167	184	14.7M	37.67
										AVERAGE	**19.1%**

Table 2.1: Comparison between the number of subproblems using Raster Points and using Discretization Points for the instances adapted from [18] and [27].

When we consider orthogonal rotations, the use of raster points also leads to a good reduction on the number of subproblems, as we can see in Table 2.2. In the average, the number of subproblems reduced to 2.13% for gcut1–gcut12 instances and to 45.44% for thpack1–thpack7 instances.

Instance	Number of items	Bin Dimensions	Raster Points m	s	u	# Subprob.	Discretization Points m	s	u	# Subprob.	# Subprob.(%) Rast./Discr.
gcut1_3d	10	(250, 250, 250)	15	15	15	3.3K	92	92	92	778.6K	0.43
gcut2_3d	20	(250, 250, 250)	41	41	41	68.9K	142	142	142	2.8M	2.41
gcut3_3d	30	(250, 250, 250)	58	58	58	195.1K	152	152	152	3.5M	5.56
gcut4_3d	50	(250, 250, 250)	81	81	81	531.4K	166	166	166	4.5M	11.62
gcut5_3d	10	(500, 500, 500)	23	23	23	12.1K	154	154	154	3.6M	0.33
gcut6_3d	20	(500, 500, 500)	28	28	28	21.9K	201	201	201	8.1M	0.27
gcut7_3d	30	(500, 500, 500)	43	43	43	79.5K	232	232	232	12.4M	0.64
gcut8_3d	50	(500, 500, 500)	95	95	95	857.3K	292	292	292	24.8M	3.44
gcut9_3d	10	(1000, 1000, 1000)	17	17	17	4.9K	174	174	174	5.2M	0.09
gcut10_3d	20	(1000, 1000, 1000)	32	32	32	32.7K	294	294	294	25.4M	0.13
gcut11_3d	30	(1000, 1000, 1000)	60	60	60	216K	461	461	461	97.9M	0.22
gcut12_3d	50	(1000, 1000, 1000)	85	85	85	614.1K	511	511	511	133.4M	0.46
										AVERAGE	**2.13%**
thpack1	3	(587, 233, 220)	393	58	50	1.1M	490	137	124	8.3M	13.69
thpack2	5	(587, 233, 220)	451	99	86	3.8M	519	165	152	13M	29.5
thpack3	8	(587, 233, 220)	486	132	119	7.6M	537	183	170	16.7M	45.7
thpack4	10	(587, 233, 220)	496	142	129	9M	542	188	175	17.8M	50.95
thpack5	12	(587, 233, 220)	504	150	137	10.3M	546	192	179	18.7M	55.19
thpack6	15	(587, 233, 220)	511	157	144	11.5M	549	195	182	19.4M	59.29
thpack7	20	(587, 233, 220)	520	166	153	13.2M	554	200	187	20.7M	63.74
										AVERAGE	**45.44%**

Table 2.2: Comparison between the number of subproblems using Raster Points and using Discretization Points for the instances adapted from [18] and [27], considering rotations.

2.6.2 Results for the Unconstrained Knapsack problem

In this section, we present the computational results for the 3D Unconstrained Knapsack problem. For this section, we consider the value v_i of each box i equal to its volume. Note that for the thpack instances the values in each group correspond to the average volume for that group. We first observe that for all instances, the computational time required to solve each instance was less than 0.001 second.

The columns of the Table 2.3 have the following information: Instance name, Volume for the case without rotations, Volume for the case with rotations, percentage of volume increased when considering rotations, Volume for the 4-staged case without rotations, Volume for the 4-staged case with rotations, percentage of volume increased when considering rotations in 4-staged patterns.

As one would expect, we have a better use of the bin when orthogonal rotations are allowed. Indeed, when we compare the occupied volume of the bin in Table 2.3, the use of rotations leads to an improvement of 5.63% on gcut instances and of 3.19% on thpack instances, on average. When considering 4-staged patterns, the use of rotations leads to an improvement of 6.65% on gcut instances and of 3.89% on thpack instances, on average.

Instance	Unconstrained 3DK			4-Staged Unconstrained 3DK		
	Without Rot.	With Rot.	Increase (%)	Without Rot.	With Rot.	Increase (%)
gcut1_3d	80.6	86.7	7.58	80.6	85.7	6.4
gcut2_3d	84.9	94.9	11.82	84.4	93.1	10.36
gcut3_3d	92.5	95.3	3	88.1	94.4	7.19
gcut4_3d	95.4	97	1.63	91.6	96.7	5.62
gcut5_3d	84.3	94.1	11.52	83.8	92.5	10.34
gcut6_3d	84.8	90	6.04	81.8	88	7.54
gcut7_3d	88.1	93.3	5.95	87.6	93.1	6.34
gcut8_3d	93.2	96.6	3.67	92.6	96.6	4.4
gcut9_3d	93.2	96.5	3.59	93.2	96.5	3.59
gcut10_3d	85.2	89	4.51	85.2	89	4.51
gcut11_3d	91.4	95	3.88	89.2	95	6.49
gcut12_3d	92.7	96.7	4.39	89.7	96	7.04
		AVERAGE	**5.63%**			**6.65%**
thpack1	90.9	98.1	7.94	89.5	97	8.35
thpack2	94.4	98.9	4.73	93	98.1	5.45
thpack3	96.7	99.3	2.74	95.4	98.7	3.5
thpack4	97.3	99.5	2.22	96	99	3.06
thpack5	97.7	99.6	1.88	96.5	99.1	2.67
thpack6	98.2	99.7	1.55	97.1	99.3	2.28
thpack7	98.6	99.8	1.25	97.6	99.5	1.93
		AVERAGE	**3.19%**			**3.89%**

Table 2.3: Results for the 3KPG problem on instances adapted from [18] and [27].

2.6.3 Results for the Cutting Stock problem

The results for the 3CSG problem and its variants are shown in Tables $2.4, \ldots, 2.7$. For each of them, we indicate the instance name; a lower bound (LB) for the value of an optimum integer solution (obtained by solving the linear relaxation (2.4) by the algorithm Simplex$_{CS}$); the difference (in percentage) between the solutions obtained by the algorithm CG3CSG and the lower bound (LB); the CPU time in seconds; the total number of columns generated; the solution obtained only by the algorithm HFF3 (or HFF3^r); and the difference between (improvement over) HFF3 (respectively, HFF3^r) and algorithm CG3CSG (respectively, CG3CSG^r).

We exhibit in Table 2.4 and Table 2.5 the results for the non-staged cutting stock problem.

In these tables we can see that the difference between the solutions of the algorithm CG3CSG (and CG3CSG^r) and the lower bound (LB) is 0.407% (and 2.320%), on average. When we compare the performance of the column generation algorithm with the algorithm HFF3 (respectively, HFF3^r) the improvement on the value of the solution is of 20.932% (respectively, 29.819%), on average. The time spent to solve these instances was at most 42 seconds for the 3CSG problem and at most 2600 seconds for the 3CSG^r problem. For the k-staged version, $k = 4$, we show in Table 2.6 and Table 2.7 the results obtained. We omitted the results for $k = 3$, since they are very similar to those for $k = 4$.

Instance	Solution of CG3CSG	LB	Difference from LB (%)	Time (s)	Columns generated	HFF3	Improvement over HFF3 (%)
gcut1_3d	177	177	*0.000%*	0.03	72	181	2.21%
gcut2_3d	220	220	*0.000%*	0.56	652	245	10.20%
gcut3_3d	142	140	1.429%	6.93	2,272	194	26.80%
gcut4_3d	520	517	0.580%	41.41	4,230	747	30.39%
gcut5_3d	122	122	*0.000%*	0.03	48	160	23.75%
gcut6_3d	305	304	0.329%	0.20	338	364	16.21%
gcut7_3d	395	394	0.254%	0.66	607	467	15.42%
gcut8_3d	371	369	0.542%	26.87	3,610	558	33.51%
gcut9_3d	60	60	*0.000%*	0.08	156	70	14.29%
gcut10_3d	217	216	0.463%	0.08	150	276	21.38%
gcut11_3d	191	189	1.058%	1.33	958	281	32.03%
gcut12_3d	429	428	0.234%	8.63	1,473	572	25.00%
		AVERAGE	**0.407%**				**20.932%**

Table 2.4: Results for the 3CSG problem on instances adapted from [27].

Instance	Solution of CG3CSG^r	LB	Difference from LB (%)	Time (s)	Columns generated	HFF3	Improvement over HFF3 (%)
gcut1_3d	163	161	1.242%	0.15	150	181	9.94%
gcut2_3d	157	153	2.614%	4.63	466	255	38.43%
gcut3_3d	135	129	4.651%	154.09	2,977	199	32.16%
gcut4_3d	460	453	1.545%	518.62	3,669	666	30.93%
gcut5_3d	100	98	2.041%	0.31	119	140	28.57%
gcut6_3d	226	225	0.444%	2.07	438	330	31.52%
gcut7_3d	372	369	0.813%	17.52	1,032	467	20.34%
gcut8_3d	327	318	2.830%	2,554.40	8,258	529	38.19%
gcut9_3d	57	54	5.556%	0.25	187	81	29.63%
gcut10_3d	198	196	1.020%	1.66	226	269	26.39%
gcut11_3d	167	161	3.727%	136.24	2,432	282	40.78%
gcut12_3d	375	370	1.351%	525.99	3,580	543	30.94%
		AVERAGE	**2.320%**				**29.819%**

Table 2.5: Results for the 3CSG^r problem on instances adapted from [27].

Observing Table 2.6 and Table 2.7, we have a difference of 0.381% (and 2.402%), on average, between the values of the solutions found by the algorithm CG3CSG (and CG3CSG^r) and the lower bound (LB). Moreover, comparing them with the HFF3 (respectively, HFF3^r) the gain in the value of the solution was 19.088% (respectively, 29.694%), on average.

The algorithm CG3CSG found an optimum solution for the instances gcut1_3d, gcut2_3d, gcut5_3d, gcut9_3d as shown in Table 2.4 and Table 2.6.

Instance	Solution of CG3CSG	LB	Difference from LB (%)	Time (s)	Columns generated	HFF3	Improvement over HFF3 (%)
gcut1_3d	177	177	*0.000%*	0.05	106	181	2.21%
gcut2_3d	220	220	*0.000%*	0.40	428	245	10.20%
gcut3_3d	146	144	1.389%	8.15	2,103	194	24.74%
gcut4_3d	519	517	0.387%	39.04	3,906	747	30.52%
gcut5_3d	132	132	*0.000%*	0.03	62	160	17.50%
gcut6_3d	305	304	0.329%	0.06	120	364	16.21%
gcut7_3d	396	394	0.508%	0.56	511	467	15.20%
gcut8_3d	399	397	0.504%	28.84	3,243	558	28.49%
gcut9_3d	62	62	*0.000%*	0.05	91	70	11.43%
gcut10_3d	218	217	0.461%	0.10	157	276	21.01%
gcut11_3d	204	202	0.990%	2.31	1,198	281	27.40%
gcut12_3d	434	434	*0.000%*	14.52	1,806	572	24.13%
		AVERAGE	**0.381%**				**19.088%**

Table 2.6: Results for the *4-staged* 3CSG problem on instances adapted from [27].

Instance	Solution of CG3CSG^r	LB	Difference from LB (%)	Time (s)	Columns generated	HFF3	Improvement over HFF3 (%)
gcut1_3d	163	161	1.242%	0.13	118	181	9.94%
gcut2_3d	157	153	2.614%	5.34	459	255	38.43%
gcut3_3d	136	130	4.615%	121.33	2,697	199	31.66%
gcut4_3d	460	453	1.545%	650.82	3,883	666	30.93%
gcut5_3d	100	98	2.041%	0.35	133	140	28.57%
gcut6_3d	228	225	1.333%	3.51	601	330	30.91%
gcut7_3d	373	369	1.084%	17.34	908	467	20.13%
gcut8_3d	325	319	1.881%	2,155.94	7,789	529	38.56%
gcut9_3d	57	54	5.556%	0.28	194	81	29.63%
gcut10_3d	198	196	1.020%	1.24	177	269	26.39%
gcut11_3d	167	161	3.727%	158.81	2,657	282	40.78%
gcut12_3d	378	370	2.162%	920.53	4,376	543	30.39%
		AVERAGE	**2.402%**				**29.694%**

Table 2.7: Results for the *4-staged* 3CSG^r problem on instances adapted from [27].

2.6.4 Results for the 3CSVG problem

We tested the algorithm CG3CSVG (and CG3CSVG^r) with the instances above mentioned, with three different bins. In these instances, the value of each bin corresponds to its volume. The results are shown in Table 2.8 and Table 2.9.

We can note that the problem with different bins size is harder to solve, demanding more computational time than the 3CSG problem. But the results were also very good, where the largest difference from the lower bound for the 3CSVG (3CSVG^r) problem was 2.052% (7.907%), and was 1.260% (4.196%), on average.

Instance	Solution of CG3CSVG	LB	Difference from LB (%)	Time (s)	Columns generated
gcut1_3d	2,431,875,000	2,415,000,000.0	0.699%	0.60	1,821
gcut2_3d	2,386,093,750	2,338,125,000.0	2.052%	9.64	10,699
gcut3_3d	2,179,687,500	2,137,243,406.8	1.986%	114.06	33,936
gcut4_3d	6,894,218,750	6,845,773,809.5	0.708%	1,572.03	163,072
gcut5_3d	13,342,500,000	13,190,833,333.3	1.150%	0.35	542
gcut6_3d	29,420,000,000	29,130,171,875.0	0.995%	5.73	7,796
gcut7_3d	36,553,750,000	36,153,136,160.7	1.108%	44.05	26,332
gcut8_3d	41,788,750,000	41,280,158,270.4	1.232%	1,622.03	197,427
gcut9_3d	59,860,000,000	58,847,226,277.4	1.721%	0.41	646
gcut10_3d	197,420,000,000	196,062,395,833.3	0.692%	0.33	550
gcut11_3d	174,270,000,000	171,061,388,146.2	1.876%	74.32	38,689
gcut12_3d	370,100,000,000	366,802,923,728.8	0.899%	143.49	18,204
		AVERAGE	**1.260%**		

Table 2.8: Results for the 3CSVG problem on instances adapted from [27].

Instance	Solution of CG3CSVG^r	LB	Difference from LB (%)	Time (s)	Columns generated
gcut1_3d	1,582,187,500	1,521,787,500.0	3.969%	5.56	3,946
gcut2_3d	1,917,812,500	1,823,075,945.0	5.197%	169.04	12,384
gcut3_3d	2,011,718,750	1,908,532,902.9	5.407%	4,049.10	68,502
gcut4_3d	5,819,531,250	5,652,226,962.2	2.960%	113,341.67	373,377
gcut5_3d	10,518,750,000	9,932,319,046.0	5.904%	14.35	4,639
gcut6_3d	22,545,000,000	21,728,068,481.4	3.760%	178.02	20,807
gcut7_3d	31,166,250,000	30,536,088,859.9	2.064%	2,464.12	77,720
gcut8_3d	38,084,200,000	37,119,105,733.3	2.534%	335,101.12	354,237
gcut9_3d	54,280,000,000	50,302,713,615.5	7.907%	9.61	6,112
gcut10_3d	157,500,000,000	154,562,209,821.4	1.901%	64.77	6,583
gcut11_3d	152,710,000,000	143,306,761,029.1	6.562%	7,636.69	102,188
gcut12_3d	300,410,000,000	293,985,781,261.7	2.185%	51,168.44	211,262
		AVERAGE	**4.196%**		

Table 2.9: Results for the 3CSVG^r problem on instances adapted from [27].

Table 2.10 and Table 2.11 show the results for the staged version of the problem. We note that some instances like gcut4_3d, gcut8_3d and gcut12_3d require tens of thousand of seconds to be solved. On the other hand, when we compare the solutions found by the algorithm CG3CSVG (and CG3CSVG^r) and the lower bound, the difference is 0.970% (and 3.920%), on average.

Instance	Solution of CG3CSVG	LB	Difference from LB (%)	Time (s)	Columns generated
gcut1_3d	2,432,500,000	2,415,000,000.0	0.725%	0.43	1,193
gcut2_3d	2,443,125,000	2,417,773,437.5	1.049%	8.85	8,907
gcut3_3d	2,238,750,000	2,205,086,568.8	1.527%	149.56	39,142
gcut4_3d	6,963,906,250	6,900,824,728.3	0.914%	1,678.22	134,327
gcut5_3d	14,187,500,000	14,122,500,000.0	0.460%	0.25	356
gcut6_3d	29,960,000,000	29,722,351,562.5	0.800%	4.88	5,936
gcut7_3d	37,412,500,000	37,028,616,071.4	1.037%	33.46	18,482
gcut8_3d	43,150,000,000	42,814,590,460.7	0.783%	958.29	90,063
gcut9_3d	61,620,000,000	61,051,648,936.2	0.931%	0.17	271
gcut10_3d	198,200,000,000	196,451,666,666.7	0.890%	1.17	1,532
gcut11_3d	181,930,000,000	178,705,312,500.0	1.804%	24.25	12,067
gcut12_3d	374,660,000,000	371,975,610,351.6	0.722%	258.72	20,801
		AVERAGE	**0.970%**		

Table 2.10: Results for the *4-staged* 3CSVG problem on instances adapted from [27].

Instance	Solution of CG3CSVG^r	LB	Difference from LB (%)	Time (s)	Columns generated
gcut1_3d	1,597,500,000	1,551,045,372.6	2.995%	3.18	3,059
gcut2_3d	1,969,062,500	1,871,147,927.3	5.233%	253.32	14,080
gcut3_3d	2,051,406,250	1,948,098,696.7	5.303%	4,481.70	79,799
gcut4_3d	5,924,218,750	5,756,335,128.3	2.917%	67,161.89	276,688
gcut5_3d	10,541,250,000	10,157,437,500.0	3.779%	9.88	2,506
gcut6_3d	23,266,250,000	22,752,722,529.8	2.257%	115.47	14,935
gcut7_3d	31,935,000,000	31,032,417,461.1	2.909%	2,782.32	90,438
gcut8_3d	38,182,500,000	37,219,195,377.3	2.588%	116,401.04	260,380
gcut9_3d	55,420,000,000	51,183,053,219.9	8.278%	6.19	3,217
gcut10_3d	160,710,000,000	156,510,662,983.4	2.683%	59.95	5,476
gcut11_3d	157,240,000,000	149,207,472,717.2	5.383%	5,687.61	79,269
gcut12_3d	305,530,000,000	297,446,392,419.0	2.718%	38,753.94	151,803
		AVERAGE	**3.920%**		

Table 2.11: Results for the *4-staged* 3CSVG^r problem on instances adapted from [27].

2.6.5 Results for the Strip Packing problem

The results obtained for the k-staged 3SPG and 3SPG^r problems with $k = 4$ are shown in Table 2.12 and Table 2.13. We omit the results for $k = 3$ because they were very similar to the case $k = 4$. As expected, the computational time required to solve these problems is considerably larger than the time required to solve the corresponding cutting stock problems. The instance gcut12_3d, for example, (and its version with rotation) required 461 seconds (28057 seconds) to be solved for the strip packing problem, and demanded 14 seconds (920 seconds) for the 4-staged version of the cutting stock problem. But the algorithm CG3SPG (and CG3SPG^r) ob-

tained very good results, computing solutions that differ from the lower bound at most 0.835% (and 1.534%). Moreover the improvement over M-HFF3 was 7.779% (and 22.325%), on average.

Instance	Solution of CG3SPG	LB	Difference from LB (%)	Time (s)	Columns generated	M-HFF3	Improvement over M-HFF3 (%)
gcut1_3d	35,510	35,458.2	0.146%	0.02	41	35,648	0.39%
gcut2_3d	45,400	45,372.2	0.061%	0.72	138	48,151	5.71%
gcut3_3d	37,632	37,565.5	0.177%	26.60	1,201	43,537	13.56%
gcut4_3d	112,507	112,334.5	0.154%	303.90	5,077	134,169	16.15%
gcut5_3d	54,311	54,208.8	0.188%	0.02	39	55,413	1.99%
gcut6_3d	114,387	114,114.7	0.239%	0.37	408	127,178	10.06%
gcut7_3d	162,829	162,551.2	0.171%	2.88	370	182,543	10.80%
gcut8_3d	185,854	185,425.5	0.231%	440.59	5,993	208,859	11.01%
gcut9_3d	58,804	58,317.3	0.835%	0.04	79	61,002	3.60%
gcut10_3d	191,638	190,937.9	0.367%	0.34	238	205,111	6.57%
gcut11_3d	192,456	191,962.8	0.257%	19.61	1,915	209,980	8.35%
gcut12_3d	399,664	398,647.1	0.255%	461.47	3,930	421,417	5.16%
		AVERAGE	**0.257%**				**7.779%**

Table 2.12: Results for the *4-staged* 3SPG problem on instances adapted from [27].

Instance	Solution of CG3SPG^r	LB	Difference from LB (%)	Time (s)	Columns generated	M-HFF3	Improvement over M-HFF3 (%)
gcut1_3d	24,863	24,757.1	0.428%	0.73	246	32,808	24.22%
gcut2_3d	30,824	30,440.4	1.260%	105.28	2,276	43,364	28.92%
gcut3_3d	32,246	31,922.2	1.014%	978.46	7,776	41,750	22.76%
gcut4_3d	90,838	90,175.2	0.735%	14,590.06	31,584	117,003	22.36%
gcut5_3d	40,931	40,263.0	1.659%	1.82	147	52,695	22.32%
gcut6_3d	87,297	86,758.8	0.620%	39.90	1,660	113,529	23.11%
gcut7_3d	121,259	120,707.1	0.457%	441.89	5,446	159,555	24.00%
gcut8_3d	149,917	148,765.7	0.774%	23,620.73	25,946	190,709	21.39%
gcut9_3d	52,314	51,523.6	1.534%	1.22	219	60,608	13.68%
gcut10_3d	151,532	150,583.5	0.630%	22.70	303	193,338	21.62%
gcut11_3d	150,444	149,160.8	0.860%	2,005.83	4,856	193,689	22.33%
gcut12_3d	296,361	294,843.5	0.515%	28,057.17	24,402	376,038	21.19%
		AVERAGE	**0.874%**				**22.325%**

Table 2.13: Results for the *4-staged* 3SPG^r problem on instances adapted from [27].

2.7 Concluding Remarks

We presented algorithms and computational tests for the problems 3KPG, 3CSG, 3CSVG and 3SPG and its variants with k stages and orthogonal rotations.

For the Three-dimensional Unconstrained Knapsack and its variants, the results obtained showed that the use of raster points in the dynamic programming approach was very successful, with considerable reduction on the number of subproblems. On the oriented 3DK problem with gcut instances, for example, the number of subproblems reduced to less than 0.77% of the number of subproblems using discretization points.

When orthogonal rotations are allowed, the occupied volume of the bin increases significantly (on average, this improvement was 5.63% on gcut instances). This is natural, since the domain of the feasible solutions increases too. The highlight is for the computational time, since all instances were solved (to optimality) in at most 0.01 seconds.

For the Three-dimensional Cutting Stock Problem and its variants, the column generation algorithm found solutions, on average, within 1.8% of the lower bound. And, when we compare with the primal heuristic we have high improvements. The computational time was high for the case when orthogonal rotations are allowed. We had instances solved in about 2600 seconds.

For the Three-dimensional Cutting Stock Problem with Variable Bin Size (and its variants) the column generation algorithm found solutions differing 2.6%, on the average, from the lower bound. On the other hand, a lot of computational time (more than 100 thousand seconds), was required to solve some instances, mainly for the case in which orthogonal rotations are allowed. So this problem showed to be harder to solve than the 3CSG problem.

The column generation algorithms for the Strip Packing problem and its variants also obtained solutions very close to the lower bound: the difference was at most 1.6%. As in the case of the 3CSG and 3CSVG problems, the improvement over the solutions returned by the primal heuristics was larger than 22.5%, on average. It is important to note that the solutions for the k-staged version of the 3CSG, 3CSVG and 3SPG problems for $k = 3$ were very similar to those for $k = 4$. The main difference was in the little increase of computational time when $k = 4$.

The computational results indicate that the algorithms proposed in this paper may be useful to solve real-world instances of moderate size. For the instances considered here, the algorithms found optimum or quasi-optimum solutions in a satisfactory amount of computational time.

Chapter 3

Approaches for the Two-Dimensional Non-Guillotine Cutting Problems

Abstract

We investigate the Two-Dimensional Unconstrained Knapsack problem and the Two-Dimensional Cutting Stock problem, both for the case in which the cuts must be orthogonal, but need not be of guillotine type. For the 2D Unconstrained Knapsack problem, we present algorithms that use the concept of reduced raster points, first order non-guillotine cuts and the known L-approach. We prove that there is no loss in the quality of the solution obtained by the L-approach if we consider only cuts on the raster points. We also show an example where the L-approach fails to obtain an optimum solution for this problem. For the 2D Cutting Stock problem, we present a column generation heuristic that uses the algorithm presented for the 2D Knapsack problem to generate columns. Integer solutions were obtained using the relaxation of the integer formulation, the reapplication of the method to residual instances and by a constructive heuristic. Computational tests on instances adapted from the literature indicate that the approaches can be useful in practice. For many instances we obtained better results than those already known.

3.1 Introduction

We present algorithms and computational results for the Unconstrained Non-Guillotine Knapsack and the Cutting Stock problems, both in the two-dimensional case. These problems have many practical applications, as for example, in loading boxes into pallets, loading pallets into trucks, and the cutting of large objects to produce small one as in the cutting of glass, steel, wood or paper. These problems, known to be NP-hard [37], can be defined as follows.

TWO-DIMENSIONAL UNCONSTRAINED KNAPSACK PROBLEM (2KP): An instance of this problem consists of a rectangular bin $B = (L, W)$ and a list T of n types of items, each type i with dimension (l_i, w_i) and value v_i, $i = 1, \ldots, n$. The objective is to determine how to cut B into items of some of the types in T so as to maximize the total value of the items produced. Here, no bound is imposed on the number of items of each type that can be produced (some types may not occur), hence the term *unconstrained*. An instance of this problem is denoted by a tuple (L, W, l, w, v), where $l = (l_1, \ldots, l_n)$ and w and v are lists defined likewise.

TWO-DIMENSIONAL CUTTING STOCK PROBLEM (2CS): Given an unlimited quantity of rectangular bins $B = (L, W)$ and n types of items, each type i with dimension (l_i, w_i) and demand d_i, $i = 1, \ldots, n$, determine how to cut the smallest number of bins B so as to produce d_i units of each item type i. An instance for this problem is given by a tuple (L, W, l, w, d).

We also study the case where the items can be rotated orthogonally: the corresponding problems are denoted 2KP^r and 2CS^r, respectively. In these problems an item i of dimension (l_i, w_i) can be considered as being an item with dimension (w_i, l_i), if $w_i \leq L$ e $l_i \leq W$.

Note that, we may consider 2KP and 2KP^r as a packing problems (in this case the objective is to pack items accordingly so as to maximize the total cost). Depending on the application, one of the terms cutting or packing may be more appropriate. We shall mostly refer to cutting.

Each possible way of cutting a rectangle (bin) is called a *cutting pattern* (or simply, *pattern*). We consider here only *orthogonal cuts*: these are cuts that are orthogonal to one of the edges of the rectangle. A *guillotine cut* is an orthogonal cut that goes from one edge of the rectangle to the opposite one, parallel to the remaining edges. A *guillotine pattern* is a pattern obtained by a series of guillotine cuts applied to the original bin and to the subsequent small bins that are obtained after each cut. A *non-guillotine pattern* is a pattern that is obtained by a series of orthogonal cuts which are not necessarily of guillotine type.

In the literature there are many works on guillotine cuts [8, 27, 41, 42, 48]. But for the unconstrained non-guillotine problem only few results can be found, as mentioned by Hadjiconstantinou and Iori (2007) [46].

In this paper we focus the mentioned problems in the more general setting in which non-guillotine cutting patterns are considered. From now on, we refer to these problems as TWO-DIMENSIONAL UNCONSTRAINED NON-GUILLOTINE KNAPSACK PROBLEM (2KPNG) and TWO-DIMENSIONAL NON-GUILLOTINE CUTTING STOCK PROBLEM (2CSNG). The corresponding versions in which orthogonal rotations are allowed are denoted as 2KPNG^r and 2CSNG^r, respectively.

To specify the position and the direction of the cuts in the algorithms, we use the Cartesian plane $\mathbb{R}^2$ with the x-coordinate and the y-coordinate representing the length and the width, respectively. The positions $(0, 0)$ and (L, W) represent, respectively, the bottom left corner and the top right corner of a bin with dimension (L, W). We adopt the convention that the position

of an item is specified by its bottom left corner. We also assume, without loss of generality, that the values in each instance are all integer numbers and that the cuts are infinitely thin.

3.1.1 Literature Review

The unconstrained guillotine knapsack problems, in the one- and two-dimensional cases, have been investigated since the sixties. In 1972, Herz [48] presented, for the two-dimensional version, a recursive approach that uses the so-called *discretization points* to obtain canonical patterns.

Cintra *et al.* (2008) [27] presented dynamic programming algorithms for the 2KP and 2KP^r problems considering guillotine patterns (these problems were denoted by 2KPG and 2KPG^r, respectively). They solved optimally instances of the OR-Library [10] spending low CPU time (finding solutions for some unsolved instances).

For the constrained non-guillotine problem, Beasley (1985) [8] proposed in 1985 a branch and bound approach in which the upper bound is calculated by a Lagrangean relaxation. Fekete and Schepers (2004) [36] showed a two-level tree search algorithm for solving the d-dimensional version of this problem.

Arenales and Morabito (1995) [6] developed an AND/OR graph approach and a branch and bound search for the 2KPNG and 2KPNG^r problems. An interesting variant of the 2KPNG^r problem is the Manufacturer's Pallet Loading problem (MPL). In this problem, given the dimensions of a bin and of an item (with unlimited copies), the objective is to pack/cut the largest possible number of the given items.

A heuristic procedure to solve the MPL problem was presented by Morabito and Morales (1998) [74]. This heuristic, called *Recursive Five-block Heuristic*, divides recursively the (rectangular) bin into five (or less) smaller (rectangular) bins using the so-called *first-order non-guillotine cuts*.

An efficient algorithm called L-approach was proposed by Lins *et al.* (2003) [65] to solve the MPL. They showed that this approach improves the solution returned by the five-block heuristic and conjectured that it always finds optimum solutions for the MPL problem when the bin is rectangular.

Recently, Birgin *et al.* (2010) [17] presented a refined version of the recursive five-block heuristic. Moreover, the authors also proved some theoretical results and made use of the *reduced raster points* (simply called *raster points* from now on) of Scheithauer (1997) [83] combined with their algorithms. They left an open question about the L patterns: whether it is possible to generated all of them using only raster points (instead of the whole set of discretization points).

The first column generation approaches for the one- and two-dimensional Cutting Stock problem restricted to guillotine cuts, called 1CSG and 2CSG, were presented by the semi-

nar works of Gilmore and Gomory (1961; 1963; 1965) [41, 42, 43]. Alvarez-Valdes *et al.* (2002) [4] also investigated the 2CSG problem, for which they presented a column generation based algorithm that uses the recurrence formulas described in Beasley (1985) [8].

Cintra *et al.* (2008) [27] also considered the 2CSG problem, and presented a heuristic based on the column generation approach that uses the exact algorithm mentioned previously for the guillotine knapsack problem to generate new columns (see Section 3.2.2 for details).

3.1.2 Contributions

We propose recursive and dynamic programming algorithms for the 2KPNG and 2KPNG^r problems. In particular, we extend the algorithms proposed for the MPL problem presented in Birgin *et al.* (2010) [17], developing a new version of the recursive five-block heuristic and of the L-approach.

Computational experiments showed that the L-approach gives good results, but sometimes consumes long CPU time. So, we use this approach limiting the number of recursive calls to a fixed parameter k (at most 7). We refer to this strategy as $L^{(k)}$-approach. The lower bound is computed by the exact dynamic programming algorithm proposed for 2KPG and 2KPG^r problems in [27].

We prove the conjecture of Birgin *et al.* (2010) [17] that L patterns can be generated using only raster points. Using this result, for many instances we noted that the number of subproblems solved by the L-approach were drastically reduced.

We present an instance for the unconstrained non-guillotine knapsack problem for which the L-approach (with cuts done on raster points) fails to compute an optimum solution. This proves that the L-approach may not find an optimum unconstrained non-guillotine pattern.

For the 2CSNG problem, we present a column generation based heuristic derived from the work of Cintra *et al.* (2008) [27] The integer solutions were calculated by first solving the relaxation of an integer linear formulation; and then by solving the residual instances with the reapplication of the method and a constructive heuristic. We also use the L-approach to generate new columns and to compute a lower bound.

The computational results reported here show that the presented algorithms find optimum solutions or improve the best solutions that were known for almost all instances considered. These results show that the proposed algorithms outperform the other heuristics and exact algorithms we found in the literature.

This paper is organized as follows. In Section 3.2, we define raster points and discuss how we obtain lower bounds. In Section 3.3.2, we present the L-approach and prove some theoretical results. Section 3.3 presents the algorithms for the 2KPNG and 2KPNG^r. The algorithms for 2CSNG and 2CSNG^r problems are discussed in Section 3.4. Section 3.5 reports computational experiments for a wide range of instances. Finally, in Section 3.6 some concluding remarks and

perspectives for future research are discussed.

3.2 Some Considerations

We first define the concept of raster points, proposed by Scheithauer (1997) [83], and their generation. We observe that the algorithms presented here generate cutting patterns using only raster points. Afterwards, we discuss how we compute lower bounds for non-guillotine patterns.

3.2.1 Raster Points

A *discretization point* (of Herz [48]) is a position on the bin (under consideration) where an item may be packed (or the bin may be cut to produce an item): it always indicate a possible position for the left corner of an item. Scheithauer (1997) refined this concept introducing the idea of raster points (they form a subset of the set of the discretization points).

Let (L, W, l, w, v) be an instance for the 2KPG problem. A *discretization point of the length* (resp., of the *width*) is a value $i \leq L$ (resp., $j \leq W$) obtained by a non-negative integer linear combination of $l = (l_1, \ldots, l_n)$ (resp., $w = (w_1, \ldots, w_n)$). The set of all discretization points of the length and width are denoted by P and Q, respectively.

From the sets P and Q, we compute the sets of raster points $\tilde{P}$ (relative to P) and $\tilde{Q}$ (relative to Q) as follows:

$$\begin{array}{ll} \tilde{P} := \{\langle L - r\rangle_P \mid r \in P\}; & \text{where } \langle s\rangle_P = \max\{t \in P \mid t \leq s\}; \\ \tilde{Q} := \{\langle W - u\rangle_Q \mid u \in Q\}; & \text{where } \langle a\rangle_Q = \max\{b \in Q \mid b \leq a\}. \end{array} \tag{3.1}$$

In 2010, Birgin *et al.* [17] proved —for the MPL problem— that each pattern generated over the discretization points may be transformed into an equivalent pattern generated over the raster points.

To find the sets of raster points, we use the algorithm RRP. This algorithm finds the discretization points using the dynamic programming algorithm DDP (*Discretization using Dynamic Programming*) shown in [27]; then, it selects the raster points using (3.1). The algorithm RRP has worst-case time complexity $O(nD)$ (pseudo-polynomial), where $D := \max\{L, W\}$.

Empirical studies carried out by Birgin *et al.* (2010) and Queiroz *et al.* (2009) [17, 80] show that there is a speed up of 50% using the set of raster points instead of the discretization points.

3.2.2 Lower Bound for the Knapsack Problem

Lower bounds for the optimum values are computed by the exact dynamic programming algorithm developed by Cintra *et al.* (2008) [27] for the unconstrained guillotine case. This exact

algorithm solves the recurrence formula given by Beasley [8] considering cutting patterns generated over the discretization points. We consider the same algorithm of [27], but restricted to patterns generated only over raster points.

For rational numbers $x_r \leq L$ and $y_r \leq W$, let

$$\begin{array}{l} p(x_r) = \max\{i |\ i \in \tilde{P},\ i \leq x_r\} \text{ and} \\ q(y_r) = \max\{j |\ j \in \tilde{Q},\ j \leq y_r\}. \end{array} \tag{3.2}$$

Using the functions defined above, we have the following recurrence formula that computes the value $G(l^*, w^*)$ of an optimum guillotine solution for a bin of dimensions (l^*, w^*):

$$G(l^*, w^*) := \max \left\{ \begin{array}{l} g(l^*, w^*); \\ \max(\{G(l', w^*) + G(p(l^* - l'), w^*) |\ \ l' \in \tilde{P},\ \ l' \leq l^*/2\}); \\ \max(\{G(l^*, w') + G(l^*, q(w^* - w'))|\ \ w' \in \tilde{Q},\ \ w' \leq w^*/2\}); \end{array} \right\}. \tag{3.3}$$

where $g(l^*, w^*)$ denotes the value of the most valuable item that can be cut in a rectangle with dimension (l^*, w^*). This value is 0, if no item can be cut in such a rectangle.

We denote by DP2KPG (resp. DP2KPG^r) the dynamic programming algorithm, presented in [27], that solves the Eq. (3.3) for an instance $I = (L, W, l, w, v)$ of the 2KPG (resp. 2KPG^r) problem. As the raster points are computed first by the algorithm RRP, the worst-case time complexity of the algorithms DP2KPG and DP2KPG^r is $O(|\tilde{P}|^2|\tilde{Q}| + |\tilde{P}||\tilde{Q}|^2)$, and the space complexity is $O(L + W + |\tilde{P}||\tilde{Q}|)$.

3.3 2KPNG Problem

We discuss now the approaches we use to solve the problems 2KPNG and 2KPNG^r.

3.3.1 The Recursive Five-block Heuristic

This heuristic partitions recursively a rectangle using the so-called *first-order non-guillotine cuts*. We say that a cut is *first-order*, if it produces 5 new smaller rectangles organized in such a way that they do not configure a guillotine pattern. A *first-order non-guillotine cutting pattern* is a pattern obtained by successive guillotine and/or first-order non-guillotine cuts.

A first-order non-guillotine cut is defined by a quadruple (x_1, x_2, y_1, y_2), such that $0 \leq x_1 \leq x_2 \leq L$ e $0 \leq y_1 \leq y_2 \leq W$ (see Fig. 3.1).

There are some refinements that we may consider in this heuristic to avoid solving the same subproblem more than once. We consider the same improvements presented by Birgin *et al.* [17] to improve such heuristic.

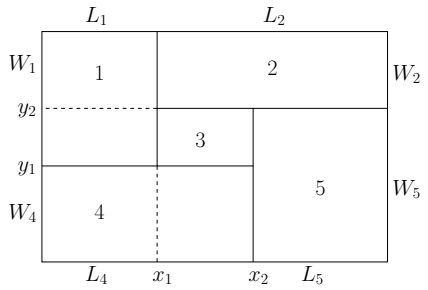

Figure 3.1: The five rectangular bins obtained after a first-order non-guillotine cut be applied.

The first improvement is related to the statement that first-order non-guillotine cutting patterns can be generated using only the set of raster points. That is, with $x_1, x_2 \in \tilde{P}$ and $y_1, y_2 \in \tilde{Q}$. Unfortunatley, this heuristic cannot find all (optimum) non-guillotine patterns.

We may also avoid equivalent patterns considering some symmetries related to the cuts. Following the results proved in [17], we can consider only first-order non-guillotine cuts that generate exactly 5 new smaller rectangles. Consider a rectangular bin of dimension (L, W) divided into 4 regions defined by 2 orthogonal straight lines: one in the position $L/2$ and the other in the position $W/2$. We call these regions A, B, C and D, as shown in Fig. 3.2.

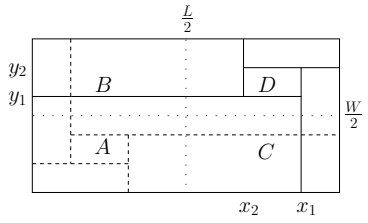

Figure 3.2: Regions used to compute the symmetries of first-order non-guillotine cuts.

Birgin *et al.* (2010) presented equivalent patterns for the first-order non-guillotine cuts. According to Fig. 3.2, first-order non-guillotine cuts into region A are symmetric to those ones in region D. The same holds for regions B and C. Other details and the proof of the mentioned improvements can be found in [17].

The algorithms Five-Block and Heur-5BL, named as Algorithm 3.1 and Algoritm 3.2, present the Recursive Five-Block heuristic used to solve the 2KPNG problem. Algorithm Five-Block uses the algorithms RRP and DP2KPG previously discussed, preparing the lower bounds and data for the recursive Algorithm Heur-5BL.

First note that when the subroutine Heur-5BL is called for a bin (L, W), we already have the

Algorithm 3.1: Five-block heuristic for the 2KPNG problem.

Input : An instance $I = (L, W, l, w, v)$ of 2KPNG problem.
Output : A solution for I.

3.1.1 $\tilde{P} \leftarrow RRP(L, l)$; $\tilde{Q} \leftarrow RRP(W, w)$
3.1.2 $LB[\,] \leftarrow$ DP2KPG$(I, \tilde{P}, \tilde{Q})$
3.1.3 $solved[i, j] \leftarrow false$ for each i, j such that $0 < i \leq L$ and $0 < j \leq W$.
3.1.4 $sol \leftarrow$ Heur-5BL(I, LB)
3.1.5 **return** sol

Algorithm 3.2: Subroutine Heur-5BL.

Input : An instance $I' = (L, W, l, w, v)$; the lower bound $LB[\,]$.
Output : A solution for I'.

3.2.1 **if** $solve[L, W] = false$ **then**
3.2.2 Build sets $\tilde{P}_L$ and $\tilde{Q}_W$ for I considering the sets $\tilde{P}$ and $\tilde{Q}$
3.2.3 **foreach** $x_1 \in \tilde{P}_L$ *such that* $x_1 \leq \lfloor \frac{L}{2} \rfloor$ **do**
3.2.4 **foreach** $x_2 \in \tilde{P}_L$ *such that* $x_1 < x_2$ *and* $x_1 + x_2 \leq L$ **do**
3.2.5 **foreach** $y_1 \in \tilde{Q}_W$ *such that* $y_1 < W$ **do**
3.2.6 **foreach** $y_2 \in \tilde{Q}_W$ *such that* $y_1 < y_2$ *and* $y_1 + y_2 \leq W$ **do**
3.2.7 **if** $x_1 + x_2 \neq L$ *or* $y_1 + y_2 \leq W$ **then**
3.2.8 Let $(L_1, W_1), \ldots, (L_5, W_5))$ be the 5 blocks (rectangles) obtained with quadruple (x_1, x_2, y_1, y_2)
3.2.9 $LB[L, W] \leftarrow \max\{LB[L, W];\ \sum_{i=1}^{5}$ Heur-5BL$(L_i, W_i, l, w, v, LB)\}$
3.2.10 **foreach** $x_1 \in \tilde{P}_L$ *such that* $x_1 \leq \lfloor \frac{L}{2} \rfloor$ **do**
3.2.11 $x_2 \leftarrow x_1$; $y_1 \leftarrow 0$; $y_2 \leftarrow 0$
3.2.12 Compute (L_1, W_1) and (L_2, W_2) blocks considering the quadruple (x_1, x_2, y_1, y_2)
3.2.13 $LB[L, W] \leftarrow \max\{LB[L, W]$; Heur-5BL(L_1, W_1, l, w, v, LB)+Heur-5BL$(L_2, W_2, l, w, v, LB)\}$
3.2.14 **foreach** $y_1 \in \tilde{Q}_W$ *such that* $y_1 \leq \lfloor \frac{W}{2} \rfloor$ **do**
3.2.15 $y_2 \leftarrow y_1$; $x_1 \leftarrow 0$; $x_2 \leftarrow 0$
3.2.16 Compute (L_2, W_2) and (L_5, W_5) considering the quadruple (x_1, x_2, y_1, y_2)
3.2.17 $LB[L, W] \leftarrow \max\{LB[L, W];$ Heur-5BL(L_2, W_2, l, w, v, LB)+Heur-5BL$(L_5, W_5, l, w, v, LB)\}$
3.2.18 $solved[L, W] \leftarrow true$
3.2.19 **return** $LB[L, W]$

optimum guillotine solution, obtained by Algorithm DP2KPG. At lines $3.2.2 - 3.2.9$ the subroutine computes a solution dividing the input rectangle (L, W) into smaller rectangles (blocks) $(L_1, W_1), \ldots, (L_5, W_5)$. All these smaller rectangles are non-empty. At lines $3.2.10 - 3.2.17$ the subroutine computes the value dividing the input rectangle (L, W) into two smaller rectangles with a guillotine cut (note that these smaller rectangles may have non-guillotine cuts). All subproblems are initially marked as not solved (meaning that it has not been solved by the Heur-5BL algorithm). Once it is solved, its status is changed, so that it is not solved again (line 3.2.18).

3.3.2 L-approach

We first discuss the central idea of the L-approach, and then we prove some theoretical results.

The main idea of the L-approach consists in generating non-guillotine patterns by partitioning recursively a rectangular (R) or L-shaped piece (L) into two new pieces, each one of which is again an R or L.

Without loss of generality, let X, Y, x and y be integer parameters. We denote by $R(X, Y)$ the rectangle whose diagonal goes from $(0, 0)$ to (X, Y). And, we denote by $L(X, Y, x, y)$, where $X \geq x \geq 0$ and $Y \geq y \geq 0$, the topological closure of $R(X, Y)$ minus the rectangle whose diagonal goes from (x, y) to (X, Y).

If $0 < x < X$ and $0 < y < Y$, then $L(X, Y, x, y)$ is said to be a *non-degenerated* L-shaped piece. When $x = X$ and $y = Y$, we have $L(X, Y, X, Y) = R(X, Y)$, a particular case of an L-shaped piece that is called a *degenerated* L. Figure 3.3 shows all possible ways to partition an L-shaped piece into two L-shaped pieces of smaller area.

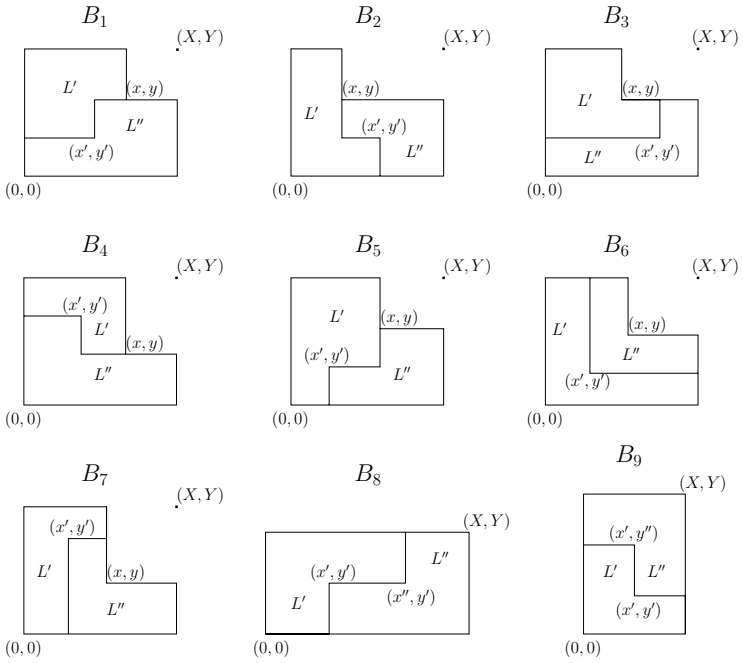

Figure 3.3: Subdivisions $B_1, \ldots, B_7$ (resp. B_8 and B_9) represent the ways we can partition a non-degenerated L-shaped piece (resp. degenerate L) into two smaller L-shaped pieces.

For a given L-shaped piece $L(X, Y, x, y)$, to indicate its partition into two smaller L-shaped pieces L' and L'' (see Fig. 3.3), we use a pair (x', y') to define the subdivisions called $B_1, \ldots, B_7$, and triples (x', x'', y') or (x', y', y'') to define the subdivisions B_8 or B_9, respectively. More details about the L-approach can be found in [17, 65].

Before presenting the recursive formula for the L-approach, we discuss and prove the open question about the raster points and the L patterns presented in [17]. This question also arises when we consider the five-block heuristic with the use of raster points. In what follows, we

prove that the L-approach can be applied using only cuts over the raster points. We first present some definitions in accordance with the terminology used by Birgin *et al.* (2010) [17].

Definition 3.1. *An $L(X, Y, x, y)$ piece is valid if its parameters belongs to the same set of points, that is, either $X, x \in P$ and $Y, y \in Q$ or $X, x \in \tilde{P}$ and $Y, y \in \tilde{Q}$.*

Definition 3.2. *Let L' and L'' be two L-shaped pieces. We say that $L' \geq L''$ if there is a transformation of L', so that for each point $p \in L''$ we also have $p \in L'$. A transformation of L' is a combination of rotation, reflection and/or translation.*

Definition 3.3. *Let C and C' be two different cuts. Let $\{L_1, L_2, \ldots, L_m\}$ and $\{L'_1, L'_2, \ldots, L'_m\}$ be the sets of pieces defined by C and C', respectively. We say that C' covers C if there is a bijection $f : \{1, 2, \ldots, m\} \to \{1, 2, \ldots, m\}$, such that $L'_{f(i)} \geq L_i$ for all $i = 1, \ldots, m$.*

Definition 3.4. *Given a set of numbers S, we define $\lceil x \rceil_S = min\{r \in S \mid x \leq r\}$.*

That is, given $x \in P$ then $\lceil x \rceil_{\tilde{P}} = min\{r \in \tilde{P} \mid x \leq r\}$. The same is valid for the set $\tilde{Q}$.

We use four lemmas obtained by Birgin *et al.* [17], described below (the proof can be found in the original work), and then we derive new results.

An important observation concerns the sets of discretization points and raster points. The set P (the same applies to sets Q, $\tilde{P}$ and $\tilde{Q}$) previously defined, is not closed under subtraction. That is, for any $a, b \in P$, where $a \leq b$, it is not necessarily true that $(b - a) \in P$. Consequently, the operator $\langle\ \rangle$ defined in Eq. (3.1) has to be used whenever some operation of subtraction is used, in order to keep *valid* any piece originated from some cut.

Lemma 3.1. *Let $x \in \tilde{P}$ and $y \in P$, where $x \geq y$. Then, $\langle x - y \rangle_P = \langle x - y \rangle_{\tilde{P}} \in \tilde{P}$.*

Lemma 3.2. *For all $x \in \tilde{P}$, we have $\langle L - \langle L - x \rangle_P \rangle_P = x$.*

Lemma 3.3. *Let $x \in P$ and $x' = \lceil x \rceil_{\tilde{P}}$. Then, $\langle L - x' \rangle_P = \langle L - x \rangle_P$.*

Lemma 3.4. *Let $x_1, x_2 \in P$, such that $x_1 \leq x_2$. Let also $x'_1 = \lceil x_1 \rceil_{\tilde{P}}$ and $x'_2 = \lceil x_2 \rceil_{\tilde{P}}$. Then, $\langle x_2 - x_1 \rangle_P \leq \langle x'_2 - x'_1 \rangle_P$.*

We first deduce new lemmas, used to prove the theorem.

Lemma 3.5. *For all $x \in P$ and $x' = \lceil x \rceil_{\tilde{P}}$, we have $\langle L - \langle L - x \rangle_P \rangle_P = \langle L - \langle L - x' \rangle_P \rangle_P = x$.*

Proof. The proof is straightforward, and follows from Lemmas 3.2 and 3.3. □

Lemma 3.6. *Let $x, y \in \tilde{P}$, where $x \geq y$. Then, $\langle x - y \rangle_P = \langle x - y \rangle_{\tilde{P}} \in \tilde{P}$.*

Proof. Note that in Lemma 3.1 we have $y \in P$. As $\tilde{P} \subseteq P$, the proof is straightforward. □

Lemma 3.7. *Let $x \in P$, $x' = \lceil x \rceil_{\tilde{P}}$ and $y \in \tilde{P}$, with $x \leq y$. Then, $\langle y - x' \rangle_P = \langle y - x \rangle_P$.*

Proof.

- First we prove (by contradiction) that $\langle y - x' \rangle_P \geq \langle y - x \rangle_P$. Suppose that we have $\langle y - x \rangle_P > \langle y - x' \rangle_P$. Letting $t = \langle y - x \rangle_P \in P$, we have $y - x' < t \leq y - x$. If we subtract y and multiply by (-1) all terms, we have $x' > y - t \geq x$. As $x \in P$, we obtain $x' > \langle y - t \rangle_P \geq x$. Since $y \in \tilde{P}$ and $t \in P$, we have by Lemma 3.1 that $r = \langle y - t \rangle_P \in \tilde{P}$. However, $x \notin \tilde{P}$. Thus, $x' > r > x$, what contradicts the definition of x';

- Now, we prove that $\langle y - x' \rangle_P \leq \langle y - x \rangle_P$. As $x' \geq x$, we have $y - x' \leq y - x$ and, thus, $\langle y - x' \rangle_P \leq \langle y - x \rangle_P$.

□

All the lemmas given above remain valid when we consider the width W and the sets Q and $\tilde{Q}$, instead of L, P and $\tilde{P}$. The following theorem is straightforward.

Theorem 3.1. *Let $\dot{L} = L(\dot{X}, \dot{Y}, \dot{x}, \dot{y})$ be an L-shaped piece, where $\dot{X}, \dot{x} \in P$ and $\dot{Y}, \dot{y} \in Q$. Then, $L' = L(\lceil \dot{X} \rceil_{\tilde{P}}, \lceil \dot{Y} \rceil_{\tilde{Q}}, \lceil \dot{x} \rceil_{\tilde{P}}, \lceil \dot{y} \rceil_{\tilde{Q}})$ defines a new piece that covers $\dot{L}$, that is, $L' \geq \dot{L}$.*

Proof. The proof is straightforward. Note that $X \geq \dot{X}$, $Y \geq \dot{Y}$, $x \geq \dot{x}$ and $y \geq \dot{y}$. □

Now, to complete the proof of the open question raised in [17], we prove the following theorem.

Theorem 3.2. *Let $L' = L(X, Y, x, y)$, where $X, x \in \tilde{P}$ and $Y, y \in \tilde{Q}$, be an L-shaped piece resulting from the application of Theorem 3.1, and let $\dot{c}$ be a cut made on the pair $(\dot{x}', \dot{y}')$ or on the triple $(\dot{x}', \dot{x}'', \dot{y}')$ or $(\dot{x}', \dot{y}', \dot{y}'')$, where $\dot{x}', \dot{x}'' \in P$ and $\dot{y}', \dot{y}'' \in Q$. Then, $(x', y') = (\lceil \dot{x}' \rceil_{\tilde{P}}, \lceil \dot{y}' \rceil_{\tilde{Q}})$ or $(x', x'', y') = (\lceil \dot{x}' \rceil_{\tilde{P}}, \lceil \dot{x}'' \rceil_{\tilde{P}}, \lceil \dot{y}' \rceil_{\tilde{Q}})$ or $(x', y', y'') = (\lceil \dot{x}' \rceil_{\tilde{P}}, \lceil \dot{y}' \rceil_{\tilde{Q}}, \lceil \dot{y}'' \rceil_{\tilde{Q}})$ define a cut c that covers $\dot{c}$, that is, $c \geq \dot{c}$.*

Proof. To prove this statement we have to consider each subdivision $B_1, \ldots, B_9$. Owing to space limitation, we just prove for the subdivision B_1. The other cases can be proved analogously.

Consider the piece $L' = L(X, Y, x, y)$ and the cut $\dot{c} = (\dot{x}', \dot{y}')$ applied on it, generating two new pieces: $\dot{L}'_1 = L(x, \langle Y - \dot{y}' \rangle_Q \, , \dot{x}', \langle Y - y \rangle_Q \,)$ and $\dot{L}'_2 = L(X, y, \langle X - \dot{x}' \rangle_P \, , y')$. On the

other hand, the cut $c = (x', y')$ applied on the piece $L' = L(X, Y, x, y)$ generates the following pieces: $L'_1 = L(x, \langle Y - y' \rangle\ , x', \langle Y - y \rangle\ )$ and $L'_2 = L(X, y, \langle X - x' \rangle\ , y')$.

We show that $L'_k \geq \dot{L}'_k$, for $k = 1, 2$ (that is, L'_k covers $\dot{L}'_k$).

To prove that $L'_1 \geq \dot{L}'_1$, note that:

- $x \in \tilde{P}$;
- $\langle Y - y' \rangle_Q \in \tilde{Q}$ by Lemma 3.6, and $\langle Y - y' \rangle_{\tilde{Q}} = \langle Y - \dot{y}' \rangle_Q$ by Lemma 3.7;
- $x' \in \tilde{P}$ and $x' \geq \dot{x}'$;
- $\langle Y - y \rangle_Q \in \tilde{Q}$ by Lemma 3.6, and $\langle Y - y \rangle_{\tilde{Q}} = \langle Y - y \rangle_Q$.

To prove that $L'_2 \geq \dot{L}'_2$, observe that:

- $X \in \tilde{P}$;
- $y \in \tilde{Q}$.
- $\langle X - x' \rangle_P \in \tilde{P}$ by Lemma 3.6, and $\langle X - x' \rangle_{\tilde{P}} = \langle X - \dot{x}' \rangle_P$ by Lemma 3.7;
- $y' \in \tilde{Q}$ and $y' \geq \dot{y}'$.

Moreover, L'_1 and L'_2 are valid pieces. □

As a corollary of the above theorem, it follows that the parameters that define each new piece obtained with a cut on the raster points also belong to the set of raster points. Therefore, cuts made in valid pieces always results in new valid pieces.

Although we had the guarantee given by the theorems above, we performed many computational experiments using not only the raster points, but also the discretization points. For all instances, the same results were obtained no matter which set of points were used. As one would expect, with the raster points the number of generated L-patterns were much smaller than the number of patterns obtained with discretization points. Some details are shown in Table 3.1 of Section 3.5.

The recurrence formula (3.4), extended from Lins *et al.* (2003) [65], is used to compute the value $v^\star$ of the solution for an $L(X, Y, x, y)$ piece. In this formula, $\mathscr{L}^k(L)$ denotes the set of all pair of pieces (L', L'') that results of the subdivision B_k, as presented in Fig. (3.3), and it considers all raster points that belong to the region delimited by such L.

$$v^\star(L) := \max \left\{ \begin{array}{l} v(L); \\ \max\limits_{1 \leq k \leq 9} \left\{ \max\{v^\star(L') + v^\star(L'') : (L', L'') \in \mathscr{L}^k(L)\} \right\} \end{array} \right\}. \tag{3.4}$$

Following Eq. (3.4), $v(L)$ corresponds to the more valuable item that can be cut in the L piece; $v(L) = 0$ means that no item can be cut in the L piece.

As we extended the L-approach to deal with the Unconstrained Knapsack problem (previously applied only to the MPL problem), the main question that arises is whether the L-approach can (or cannot) cover all non-guillotine patterns for this more general problem.

As shown in the computational tests, in Section 3.5, the L-approach could improve the value of the solutions for almost all the instances considered. However, to answer the above question, we present in Fig. 3.4 a counterexample in which the L-approach fails to find an optimum solution. This shows that the L-approach not always computes optimum (non-guillotine) solutions for the Unconstrained Knapsack problem.

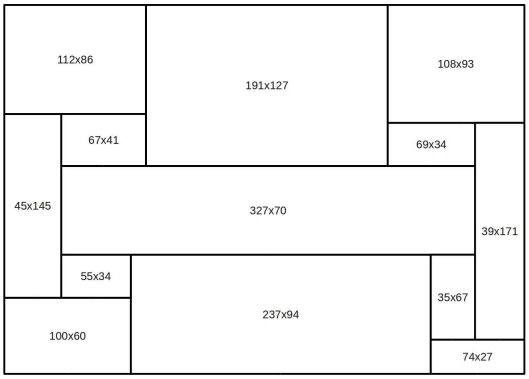

Figure 3.4: Instance in which the L-approach fails in computing the optimum solution.

For the instance presented in Fig. 3.4, the gap between the solution returned by the L-approach (value of $119{,}318$) and the optimum solution (value of $119{,}601$) is of 0.237%. The L-approach required $55{,}863.53$ seconds ($\approxeq$ 15.5 hours) of CPU time (running on the computer mentioned in Section 3.5). The five-block heuristic returned a solution (value of $119,171$) after 309 seconds and with a gap of 0.36%.

3.3.3 The $L^{(k)}$-approach

We performed some preliminary numerical experiments with the L-approach, and noticed that it spends considerable CPU time to solve some instances of the 2KPNG (mainly 2KPNG^r) problem, even with the use of raster points. This is a consequence of the depth of the recursive calls to solve the recurrence formula (3.4). In view of this, we decided to restrict the recursive calls to some fixed parameter k. We called this algorithm $L^{(k)}$-approach.

We performed further tests to see the influence of this parameter k on the value of the solution, and, of course, on the runtime to solve the instances. In the computational tests, we tried to balance the ratio: "depth limit versus CPU time". The appropriate value of k will be

discussed when dealing with the numerical experiments. As the $L^{(k)}$-approach is very similar to the L-approach, we just show the algorithm for the first one.

The main routine for the $L^{(k)}$-approach is presented in Algorithm 3.3. It uses the algorithm RRP to generate the sets of raster points, and the algorithm DP2KPG that solves the instance and returns the lower bound of the subproblems that can appear during the processing of the subroutine SolveL (line 3.3.4).

Algorithm 3.3: $L^{(k)}$-approach.

Input : An instance $I = (L, W, l, w, v)$ for the 2KPNG problem; The depth limit for the recursion calls, DEP.
Output : A solution for I.

3.3.1 $\tilde{P} \leftarrow RRP(L, l)$; $\tilde{Q} \leftarrow RRP(W, w)$
3.3.2 **foreach** *L-shaped piece* $L' = L(X, Y, x, y)$ *with* $x \leq X \leq L$ *and* $y \leq Y \leq W$ **do**
3.3.3 $\quad solved[L'] \leftarrow false$
3.3.4 $\quad LB[L'] \leftarrow 0$
3.3.5 $LB[\ ] \leftarrow \text{DP2KPG}(I, \tilde{P}, \tilde{Q})$
3.3.6 **return** *SolveL*$(R(L, W),\ DEP,\ LB[\,])$

The algorithm SolveL —see Algorithm 3.4— solves the recurrence formula (3.4) considering the depth limit imposed for the recursive calls. First, (line 3.4.1) the algorithm verifies if the subproblem $L(X, Y, x, y)$ was already solved and if it is within the allowed depth. Initially, the solution corresponds to the respective lower bound already computed. Next, (at lines $3.4.2 - 3.4.4$) the algorithm recursively computes each subdivision B_j, for $\mathscr{L}_1^j$ and $\mathscr{L}_2^j$, considering each pair (L', L'') of pieces resulting from subdivision j and with all parameters (that are raster points).

Algorithm 3.4: SolveL.

Input : The $L^* = L(X, Y, x, y)$ piece; The depth limit k; The lower bound $LB[\,]$.
Output : The value of $LB[L^*]$ and the matrix LB updated.

3.4.1 **if** $k > 0$ *and* $solved[L^*] = false$ **then**
3.4.2 $\quad$ **for** $j \leftarrow 1$ **to** 9 **do**
3.4.3 $\quad\quad$ **foreach** *pair* $(L', L'') \in \mathscr{L}^j(L)$ *obtained from the subdivision of type* B_j **do**
3.4.4 $\quad\quad\quad LB[L^*] \leftarrow \max\{LB[L^*];\ \ SolveL(L_1, k-1, LB[\,]) + SolveL(L_2, k-1, LB[\,])$
3.4.5 $\quad solved[L^*] \leftarrow true$
3.4.6 **return** $LB[L^*]$

3.3.4 Time and Space Complexity

To estimate the time and space complexity of the five-block heuristic and L-approach in the worst-case, we assume that no depth limit in the recursion is imposed. So, each subproblem

is solved at most once. The strategies were implemented by a memorized recursive algorithm. A table is used to save the information of each subproblem. As a consequence, we may affirm that the worst-case complexity of this implementation is the same of its iterative dynamic programming counterpart. For simplicity we just analyze the last one.

Let $I = (L, W, l, w, v)$ be an instance for 2KPNG problem. First we compute the sets of raster points using the algorithm RRP. As mentioned, the time complexity of this operation is $O(nL + nW)$ and the space complexity is $O(L + W)$.

To compute the lower bound we use the algorithm DP2KPG. The values obtained were saved in a table. We note that for each pair $(X, Y) \in \tilde{P} \times \tilde{Q}$ there is a subproblem whose instance is $I_{XY} = (X, Y, l, w, v)$. However, to compute the $LB[\,]$ we just call the algorithm DP2KPG with original instance $I = (L, W, l, w, v)$ as input. It is easy to see that all subproblems I_{XY} are solved when the instance I is solved by the algorithm DP2KPG, because it is exact and based on iterative dynamic programming. So, we only save informations about each subproblem I_{XY}. The time complexity to calculate the lower bound $LB[\,]$ is the same of the algorithm DP2KPG, that is: $O(|\tilde{P}|^2|\tilde{Q}| + |\tilde{P}||\tilde{Q}|^2)$. The space complexity is $O(L + W + |\tilde{P}||\tilde{Q}|)$.

The time complexity to solve only one subproblem I_{XY} by the five-block heuristic is given by: *(i)* time complexity of $O(|P| + |Q|)$ to obtain the raster points from the sets $\tilde{P}$ and $\tilde{Q}$ first computed, plus *(ii)* time complexity of $O(|\tilde{P}|^2|\tilde{Q}|^2)$ to generate all first-order non-guillotine cuts, plus *(iii)* time complexity of $O(|\tilde{P}| + |\tilde{Q}|)$ to generate all the horizontal and vertical guillotine cuts.

Consequently, the worst-case time complexity of the five-block heuristic is the time complexity of the algorithm RRP plus the time complexity to compute the lower bound $LB[\,]$ plus the time complexity to solve all possible subproblems (where we have $O(|\tilde{P}||\tilde{Q}|)$ subproblems):

Therefore, the time complexity of the five-block heuristic is given by

$$O(nL + nW + |\tilde{P}||\tilde{Q}|(|\tilde{P}|^2|\tilde{Q}|^2 + |P| + |Q|)). \tag{3.5}$$

The worst-case space complexity is clearly $O(L + W + |\tilde{P}||\tilde{Q}|)$.

Following the same idea for the L-approach, we have one subproblem I_{XY} solved with: *(i)* time complexity of $O(|P| + |Q|)$ (to get the *raster points* from the sets $\tilde{P}$ and $\tilde{Q}$ already computed) plus *(ii)* time complexity of $O(|\tilde{P}||\tilde{Q}|)$ (to generate all non-degenerated L-shaped pieces) plus *(iii)* time complexity of $O(|\tilde{P}|^2|\tilde{Q}| + |\tilde{P}||\tilde{Q}|^2)$ (to generate all degenerated L-shaped pieces).

The worst-case time complexity of the L-approach is computed by: time complexity of the algorithm RRP plus the time complexity to compute the lower bound $LB[\,]$ plus the time complexity to solve all possible subproblems. Therefore, we have a resulting time complexity of

$$O(nL + nW + |\tilde{P}|^2|\tilde{Q}|^2(|\tilde{P}||\tilde{Q}| + |P| + |Q|)), \tag{3.6}$$

where the number of subproblems is $O(|\tilde{P}|^2|\tilde{Q}|^2)$. The amount of memory required is $O(L + W + |\tilde{P}|^2|\tilde{Q}|^2)$.

It is easy to see that the algorithms proposed are pseudo-polynomial. So, they are suited for instances in which the bins have small dimensions. Also, when the bins have large dimensions but the dimensions of the items are not so small compared to the dimensions of the bin, then these algorithms have a satisfactory performance.

3.3.5 2KPNG^r Problem

Since the Knapsack problem under consideration is the unconstrained version, we can deal with the version with orthogonal rotations in a easy way, without any modification in the algorithms proposed. We just need to create another instance for the problem 2KPNG.

Given an instance I' of the 2KPNG^r problem, we create another instance I —for the 2KPNG problem— in which we add the n items and the bin B of I' to I. Then, for each item $i' \in I'$ with dimensions (l_i, w_i) and value v_i, we add to I the new item $i = (w_i, l_i, v_i)$, if it is feasible and it is not already in the instance. The same remains valid for the guillotine version.

3.4 2CSNG Problem

In this section we deal with the Two-dimensional Cutting Stock problem, where the cuts are of non-guillotine type (2CSNG). To solve this problem we use column generation based algorithms, where the columns are generated by the algorithms proposed for the unconstrained Knapsack problem.

The integer linear formulation for the Cutting Stock problem based on cutting patterns is described next. Let $\mathcal{P}$ be the set of all cutting patterns, $|\mathcal{P}| = m$; let P be a matrix of order $n \times m$, whose columns represent the cutting patterns; and let d be the vector of demands of the items. We use an integer variable x_j for each pattern $j \in \mathcal{P}$, that indicates how many times the pattern j is chosen. Thus, the following linear program is a relaxation of an integer formulation:

$$\begin{aligned} \textit{minimize} \quad & \sum_{j \in \mathcal{P}} x_j & & \\ \textit{subject to} \quad & Px & \geq & d \\ & x_j & \geq & 0 \quad \forall\, j \in \mathcal{P}. \end{aligned} \tag{3.7}$$

To solve the 2CSNG problem we use the algorithms presented in Cintra *et al.* (2008) [27]. The main idea of these algorithms is to solve the linear formulation (3.7) using the column

generation approach proposed by Gilmore and Gomory [41] in the early sixties; round down the variables to obtain a partial integer solution; and, repeat the process to deal with the residual problems. When this process cannot find a non-null solution in one iteration, a primal heuristic is used to obtain a good cutting pattern and perturb the residual instance.

In each iteration of the column generation process, for each item i we have a value y_i and we have to compute a new column (pattern) that satisfies $\sum_{i=1}^{n} y_i z_i > 1$, where z_i represents the number of times that item i appears in such pattern. Naturally, we can use any algorithm for the unconstrained (non-guillotine) Knapsack problem to compute the patterns.

The algorithm used to solve the linear program (3.7) is the algorithm SimplexCG$_2$ presented in Cintra *et al.* [27]. It corresponds to the algorithm Simplex with a column generation subroutine. More details about this algorithm can be found in [25]. In our implementation the matrix $I_{n\times n}$ corresponds to the identity matrix with n patterns, each one with items of one type and one orientation.

The subroutine to solve the 2D unconstrained non-guillotine knapsack problem, as the column generator procedure, denoted by Sol_3, is coded in the following way:

- First, we try to generate a feasible column using the algorithm DP2KPG;
- If the DP2KPG fails, we try to generate a feasible column with the five-block heuristic;
- If the five-block heuristic also fails, we try the $L^{(k)}$-approach as a last option.

Since the algorithm SimplexCG$_2$ may return a fractional solution, we also use the algorithm CG^p (presented in [27]) that receives the solution computed by SimplexCG$_2$, and returns an integer solution for the 2CSNG problem. The algorithm CG^p uses as subroutine a primal heuristic to obtain a cutting pattern that causes a perturbation of some residual instance. In this case, the primal heuristic considered is the algorithm M-HFF, that is a modified version of the heuristic HFF (*Hybrid First Fit*) to deal with demands. Other details on HFF can be found in [24].

The algorithm CG^p has exponential worst-case time complexity and halts after a finite number of iterations, since the demand is fulfilled in each iteration of the rounding process.

Although the L-approach spent considerable CPU time in solving some Knapsack instances, it could in general obtain very good results. So, we also used this algorithm as a unique algorithm to generate new columns and to calculate a lower bound for the value of an optimum integer solution (that is obtained solving the linear relaxation (3.7) by the algorithm SimplexCG$_2$).

3.4.1 2CSNG^r Problem

We use the algorithm CGR^p presented in Cintra *et al.* (2008) [27] to solve the 2CSNG^r problem. The main difference between this algorithm and the algorithm CG^p is in the subroutines called. The algorithm CGR^p make calls to the algorithms proposed for 2KPG^r and 2KPNG^r problems, that are the versions in which the items can be orthogonally rotated.

3.5 Computational Experiments

The algorithms above mentioned were implemented in $C/C++$. The tests were run in a computer with Intel® Core^{TM} 2 Quad 2.4 GHz processor, 4 GB of memory and *Linux* operating system. The linear systems in the column generation algorithms were solved by the COIN-OR CLP solver [28].

We executed the numerical experiments with many medium and large-sized instances presented in the literature. Some of the instances were proposed for others variants (constrained and/or guillotine-cut) of the Knapsack problem and some of them were not proposed for the Cutting Stock problem, but they were easily adapted for our purposes. In what follows we present the instances considered:

- Three instances, $cgcut01 - cgcut03$, from [23];
- Twelve instances, $ngcut01 - ngcut12$, from [9];
- Five instances, $m1 - m5$, from [73];
- Thirteen instances, $gcut01 - gcut13$, from [8];
- Five instances, $okp01 - okp05$, from [36];
- Twenty two instances, $uu1 - uu11,\ uw1 - uw11$, from [35];

Some of these instances are available in the OR-Library [10]. We left them publically available at the URL: `http://www.loco.ic.unicamp.br/nonguillotine2d/`.

All the above instances were used for the unconstrained Knapsack problem. Note that for this problem we just need the bin's dimensions and the items' dimensions and values, so the other information were excluded. For the instances m, $gcut$ and uw the value of each item is precisely its area.

However, only some of those instances were considered for the Cutting Stock problem due to difficulties with memory size (memory overflow) to store/solve instances with large-size dimensions. The instances $gcut01 - gcut12$ were used by Cintra *et al.* (2008) when solving the guillotine versionof the Cutting Stock problem. For the others instances we needed to generate an integer demand for each item. Each integer demand was randomly generated in the interval $[1,\ 100]$ (varying demands).

With the aim of verifying the influence of the depth limit k in the recursion calls of the $L^{(k)}$-approach, we initially executed this approach with $k = 2, \ldots, 6$. Consequently, we found solutions of lower quality spending reasonable CPU time. Best solutions, some of them equal to solutions found by L-approach, were found using $k = 7$, however the runtime increases accordingly to the increase of k. It seems to us that for large values of k the depth limit has no influence when solving the subproblems.

In order to illustrate the number of non-guillotine patterns generated by the algorithms, we present in Table 3.1 some characteristics of the instances above mentioned. In each line of this table: Instance is the instance name; (L, W) is the bin's dimension; $|\tilde{P}|$ (relative to P) and $|\tilde{Q}|$ (relative to Q) are the number of reduced raster points; Raster patterns is the number of non-guillotine subproblems using raster points; $|P|$ and $|Q|$ are the number of discretization points of Herz for the length and width, respectively; Discretization patterns is the number of non-guillotine subproblems using discretization points; Rast./Discr. is the percentage of subproblems using raster points compared to the subproblems using discretization points.

Each subproblem in the L-approach (or the first-order non-guillotine cuts in Five-block heuristic) is determined by a quadruple (X, Y, x, y) where $X, x \in \tilde{P}$ and $Y, y \in \tilde{Q}$. So the total number of subproblems (worst-case) is of magnitude $(|\tilde{P}||\tilde{Q}|)^2$.

Observing Table 3.1 the number of subproblems using raster points corresponds only to 11.59%, on average, of the number of subproblems using discretization points.

3.5.1 Results for the 2KPNG and 2KPNG^r Problems

Tables 3.2 and 3.3 show the results of the numerical experiments for the 2KPNG and 2KPNG^r problems, respectively. In each line of such tables we have: instance name; value of solution and the time (of CPU) in seconds spent to solve the respective instance (the value zero indicates that the time required is less than 0.0001s) obtained by the Five-block heuristic, $L^{(k)}$-approach (with $k = 7$) and L-approach, respectively; and, the optimum solution for the guillotine case computed by the DP2KPG (or DP2KPG^r) algorithm.

The entry "–" in the following tables represents that the solution was not returned after 10 days of processing, so we abort the processing. The solutions of the Five-block heuristic that are in boldface have the same value as the ones obtained by L-approach. The underlined solutions means that they improve the guillotine one, and they were the best result found. Note that the algorithms of this paper get equal or better results (for many instances) than the optimum guillotine solution.

As shown in Table 3.2, the algorithms (Five-block heuristic, $L^{(k)}$-approach and L-approach) obtained the same solution for 86.67% of the instances, that corresponds to 52 out of 60 instances. Only for instances $m1, uu8$ and $uu10$ the solutions obtained by the L-approach are better than the ones obtained by the Five-block heuristic. Such improvement is only of 0.13%, in the best situation (instance $m1$).

On the other hand, the Five-Block heuristic was faster than the other approaches and its CPU time was only 35.97 seconds, on average. Definitely the L-approach spent more CPU time than the other algorithms (Five-block heuristic and $L^{(k)}$-approach), since its CPU time was 7485.37 seconds, on average. The $L^{(k)}$-approach spent 288.92 seconds, on average, that is equivalent to 96.14% lesser CPU time than the L-approach.

Table 3.1: Comparisons on the number of patterns using raster points and discretization points for the orientated case.

Instance	(L, W)	n	$\lvert\tilde{P}\rvert$	$\lvert\tilde{Q}\rvert$	Raster Patterns	$\lvert P\rvert$	$\lvert Q\rvert$	Discretization Patterns	No. Patterns (%) Rast./Discr.
cgcut01	(15, 10)	7	14	11	23716	15	11	27225	87.11
cgcut02	(40, 70)	10	19	43	667489	30	57	2924100	22.83
cgcut03	(40, 70)	20	14	41	329476	27	56	2286144	14.41
ngcut01	(10, 10)	5	9	6	2916	10	8	6400	45.56
ngcut02	(10, 10)	7	11	11	14641	11	11	14641	100.00
ngcut03	(10, 10)	10	11	11	14641	11	11	14641	100.00
ngcut04	(15, 10)	5	4	11	1936	7	11	5929	32.65
ngcut05	(15, 10)	7	8	11	7744	12	11	17424	44.44
ngcut06	(15, 10)	10	14	11	23716	15	11	27225	87.11
ngcut07	(20, 20)	5	21	21	194481	21	21	194481	100.00
ngcut08	(20, 20)	7	6	21	15876	11	21	53361	29.75
ngcut09	(20, 20)	10	21	19	159201	21	20	176400	90.25
ngcut10	(30, 30)	5	31	25	600625	31	28	753424	79.72
ngcut11	(30, 30)	7	17	27	210681	24	29	484416	43.49
ngcut12	(30, 30)	10	27	31	700569	29	31	808201	86.68
m1	(100, 156)	10	23	17	152881	48	74	12616704	1.21
m2	(253, 294)	10	63	17	1147041	154	87	179506404	0.64
m3	(318, 473)	10	13	32	173056	72	156	126157824	0.14
m4	(501, 556)	10	15	15	50625	78	139	117548964	0.04
m5	(750, 806)	10	23	16	135424	124	147	332259984	0.04
gcut01	(250, 250)	10	13	5	4225	68	20	1849600	0.23
gcut02	(250, 250)	20	17	24	166464	95	112	113209600	0.15
gcut03	(250, 250)	30	44	26	1308736	143	107	234120601	0.56
gcut04	(250, 250)	50	45	50	5062500	146	146	454371856	1.11
gcut05	(500, 500)	10	10	13	16900	40	76	9241600	0.18
gcut06	(500, 500)	20	12	18	46656	96	120	132710400	0.04
gcut07	(500, 500)	30	23	19	190969	179	126	508682916	0.04
gcut08	(500, 500)	50	44	59	6739216	225	262	3475102500	0.19
gcut09	(1000, 1000)	10	15	7	11025	92	42	14930496	0.07
gcut10	(1000, 1000)	20	14	20	78400	89	155	190302025	0.04
gcut11	(1000, 1000)	30	20	38	577600	238	326	1724930448	0.03
gcut12	(1000, 1000)	50	49	42	4235364	398	363	20872736676	0.02
gcut13	(3000, 3000)	32	647	1849	1431140867809	1821	2425	19500393605625	7.34
okp1	(100, 100)	15	101	101	104060401	101	101	104060401	100.00
okp2	(100, 100)	30	101	83	70274689	101	92	86341264	81.39
okp3	(100, 100)	30	93	55	26163225	97	78	57244356	45.70
okp4	(100, 100)	33	101	95	92064025	101	98	97970404	93.97
okp5	(100, 100)	29	101	83	70274689	101	92	86341264	81.39
uu1	(500, 500)	25	29	39	1279161	171	205	1228853025	0.10
uu2	(750, 800)	30	61	38	5373124	334	322	11566572304	0.05
uu3	(1100, 1000)	25	44	37	2650384	389	317	15206095969	0.02
uu4	(1000, 1200)	38	65	83	29106025	444	570	64049486400	0.05
uu5	(1450, 1300)	50	111	116	165791376	694	643	1563426948	10.60
uu6	(2050, 1457)	38	59	50	8702500	649	495	125559921	6.93
uu7	(1465, 2024)	50	144	139	400640256	743	944	491950737664	0.08
uu8	(2000, 2000)	55	114	95	117288900	914	830	575504304400	0.02
uu9	(2500, 2460)	60	120	127	232257600	1047	1044	796744336	29.15
uu10	(3500, 3450)	55	110	177	379080900	1304	1542	1623724288	23.35
uu11	(3500, 3765)	25	2527	1156	8533479548944	3014	2461	55018623842116	15.51
uw1	(500, 500)	25	31	35	1177225	208	219	2074984704	0.06
uw2	(560, 750)	35	85	64	29593600	309	355	12032993025	0.25
uw3	(700, 650)	35	53	59	9778129	301	311	173084729	5.65
uw4	(1245, 1015)	45	115	128	216678400	623	533	110263179481	0.19
uw5	(1100, 1450)	35	61	52	10061584	452	476	46290383104	0.02
uw6	(1750, 1542)	47	98	93	83064996	764	721	303429112336	0.03
uw7	(2250, 1875)	50	93	113	110439081	905	837	573783525225	0.02
uw8	(2645, 2763)	55	119	172	418939024	1116	1263	1986712802064	0.02
uw9	(3000, 3250)	45	80	85	46240000	1082	1166	1591664838544	0.01
uw10	(3500, 3650)	60	183	130	565964100	1526	1556	5638041295936	0.01
uw11	(555, 632)	30	176	221	1512898816	366	427	24424063524	6.19

The $L^{(k)}$-approach (with $k = 7$) spent more CPU time than the Five-block heuristic and could not obtain better solutions than the latter one. With the Five-block heuristic we get better solutions for 8 instances when comparing with the $L^{(k)}$-approach approach. See in Table 3.2 the instances: $cgcut02$, $m1$, $m2$, $m5$, $gcut04$, $gcut05$, $uu4$ and $uw11$. The best improvement

Table 3.2: Results for the 2KPNG problem.

Instance	Algorithms						Optimum
	Five-block		$L^{(k)}$-approach		L-approach		Guillotine
	Value	Time (s)	Value	Time (s)	Value	Time (s)	Solution
cgcut01	**250**	0.0040	250	0.0200	250	0.1160	249
cgcut02	**3094**	0.0840	3076	1.3201	3094	5.5283	3076
cgcut03	**2240**	0.0120	2240	0.6160	2240	2.2161	2240
ngcut01	**243**	0.0000	243	0.0040	243	0.0000	243
ngcut02	**280**	0.0000	280	0.0000	280	0.0000	280
ngcut03	**268**	0.0040	268	0.0000	268	0.0000	268
ngcut04	**318**	0.0000	318	0.0040	318	0.0000	318
ngcut05	**396**	0.0000	396	0.0040	396	0.0080	396
ngcut06	**374**	0.0040	374	0.0320	374	0.0840	371
ngcut07	**1144**	0.0440	1144	0.1840	1144	1.6961	1144
ngcut08	**1039**	0.0000	1039	0.0080	1039	0.0240	1039
ngcut09	**1128**	0.0240	1128	0.1200	1128	1.5441	1128
ngcut10	**2250**	0.0200	2250	0.0000	2250	0.0000	2250
ngcut11	**2113**	0.0360	2113	0.3800	2113	1.5401	2113
ngcut12	**2039**	0.2880	2039	1.1321	2039	13.6369	2039
m1	15054	0.0080	15024	0.1040	15073	0.5440	15024
m2	**73255**	0.0520	73176	0.8040	73255	9.6766	73176
m3	**147386**	0.0080	147386	0.1640	147386	0.5240	142817
m4	**266233**	0.0040	266233	0.0440	266233	0.1280	265768
m5	**579883**	0.0120	577882	0.0920	579883	0.4840	577882
gcut01	**58480**	0.0000	58480	0.0040	58480	0.0040	56460
gcut02	**61146**	0.0080	61146	0.2520	61146	0.4400	60536
gcut03	**61275**	0.0320	61275	2.4202	61275	7.4445	61036
gcut04	**61918**	0.1240	61698	16.9931	61918	56.2635	61698
gcut05	**246000**	0.0000	246000	0.0120	246000	0.0160	246000
gcut06	**243598**	0.0040	238998	0.0520	243598	0.0840	238998
gcut07	**244306**	0.0120	244306	0.1680	244306	0.4680	242567
gcut08	**247815**	0.1720	247815	24.1975	247815	93.2858	246633
gcut09	**971100**	0.0040	971100	0.0040	971100	0.0040	971100
gcut10	**982025**	0.0080	982025	0.0680	982025	0.0960	982025
gcut11	**980096**	0.0280	980096	1.2521	980096	2.2521	980096
gcut12	**979986**	0.1400	979986	21.4053	979986	43.0907	979986
gcut13	–	–	–	–	–	–	8997780
okp1	**28513**	887.1714	28513	723.0732	28513	22991.9929	28425
okp2	**28467**	3.7682	28467	0.3200	28467	0.3120	28467
okp3	**28360**	54.2394	28360	216.9976	28360	7619.0002	28360
okp4	**458236**	666.2416	458236	752.4390	458236	51835.5115	458236
okp5	**28467**	3.8442	28467	0.2960	28467	0.3080	28467
uu1	**245205**	0.0440	245205	4.2763	245205	7.8685	242919
uu2	**595288**	0.1640	595288	13.4608	595288	65.5521	595288
uu3	**1088154**	0.1000	1088154	9.3606	1088154	28.4178	1072764
uu4	**1191071**	1.0241	1184671	174.2549	1191071	783.2930	1179050
uu5	**1870038**	4.8363	1870038	1245.4338	1870038	8456.2205	1868999
uu6	**2950760**	0.2960	2950760	17.0851	2950760	144.3610	2950760
uu7	**2943852**	11.7847	2943852	1755.9657	2943852	21516.7407	2930654
uu8	3969784	3.8242	3969784	490.0306	3970877	5438.4519	3959352
uu9	**6100692**	5.5523	6100692	1076.1513	6100692	13990.7264	6100692
uu10	11995637	11.5967	11995637	2341.7504	12001291	26252.7047	11955852
uu11	–	–	–	–	–	–	13157811
uw1	**6036**	0.0680	6036	2.2761	6036	8.4125	6036
uw2	**8468**	0.3840	8468	0.1080	8468	0.1000	8468
uw3	**6302**	0.1640	6302	0.0440	6302	0.0480	6302
uw4	**8326**	2.4362	8326	0.5480	8326	0.5840	8326
uw5	**7780**	0.4880	7780	44.1548	7780	201.6486	7780
uw6	**6615**	1.2961	6615	0.2800	6615	0.2800	6615
uw7	**10464**	1.6721	10464	0.3760	10464	0.3400	10464
uw8	**7692**	5.4123	7692	1.1481	7692	1.1001	7692
uw9	**7128**	3.1362	7128	304.7230	7128	2009.2056	7038
uw10	**7507**	7.8165	7507	1.5401	7507	1.5281	7507
uw11	**16400**	407.7535	15920	7509.1853	16400	272555.4457	15747

in solution occurs for instance $uw11$ and it was 2.93%. As observed, for high values of k the $L^{(k)}$-approach may obtain better solutions, however the CPU time will increase accordingly to.

It is worth mentioning that the values of the solutions for the guillotine case were improved for 41.67% of the instances. This improvement corresponds to at most 3.98% (instance $uw11$)

(see Table 3.2).

Next, we present in Table 3.3 the results for the case where the items can be orthogonally rotated. First of all, comparing Tables 3.2 and 3.3, we have the following increase on average on CPU time spent by the algorithms compared to the case where the items can be orthogonally rotated: 438.08% for the Five-block heuristic; 437.41% for the $L^{(k)}$-approach; and, 62.85% for the L-approach.

As one can observe in Table 3.3, the same conclusions hold for the case with rotations. Now, the algorithms found equal solution for 47 out of 60 instances, that is 78.33% of the instances. The L-approach returned better result only for instance $m1_r$, and the improvement was 0.13%. The Five-block heuristic still returned better results than the $L^{(k)}$-approach. In addition, the guillotine solution was improved 0.49%, on average, considering 28 instances.

Although the CPU time increases for the case that allows rotations of the items, of course better solutions were obtained, because the domains of the solutions also increase. In fact, when comparing the improvement (only for the L-approach) of the solutions obtained for this case and the orientated case, the difference is of 3.22%, on average.

Finally, as shown in Tables 3.2 and 3.3 we note that the L-approach obtained better solutions than the Five-block heuristic and $L^{(k)}$-approach for a small set of instances, that correspond to only 4 of a total of 120 instances. Besides that, such approach also spent much more CPU time than the other algorithms. So, the winner is the Five-block heuristic that returned the same solutions of the L-approach and $L^{(k)}$-approach for almost all instances, but spending much less CPU time.

3.5.2 Results for the 2CSNG and 2CSNG^r Problems

We show in Tables 3.4 and 3.5 the numerical experiments for 2CSNG and 2CSNG^r problems. Each column in these tables corresponds to: instance name ($Name$); value of the solution returned by the algorithm CG^p (or CGR^p) using only the L-approach to generate new columns; the lower bound (LB) for the value of an optimum integer solution obtained by solving the linear relaxation (3.7) with the algorithm CG^p (or CGR^p); the difference (in percentage) between the solutions obtained with the algorithm CG^p (or CGR^p) and the lower bound (LB). the CPU time spent in seconds when using the L-approach; the total number of columns generated ($\#Columns$); the solution obtained using the subroutine Sol_3 in the algorithm CG^p (or CGR^p); the CPU time spent in seconds when using the subroutine Sol_3; and, the difference between (improvement over) the use of subroutine Sol_3 comparing with L-approach when generating new columns. The last column shows the result when using only the algorithm DP2KPG (for guillotine version of the knapsack problem) to generate the columns.

Some of the instances previously considered by the unconstrained Knapsack problem (see Tables 3.2 and 3.3) were not considered for the Cutting Stock problem. The main reason is

Table 3.3: Results for the 2KPNG^r problem.

Instance	Algorithms						Optimum
	Five-block		$L^{(k)}$-approach		L-approach		Guillotine
	Value	Time (s)	Value	Time (s)	Value	Time (s)	Solution
cgcut01$_r$	**278**	0.0040	278	0.0400	278	0.1640	278
cgcut02$_r$	**3150**	1.2841	3150	17.1171	3150	112.3390	3147
cgcut03$_r$	**2320**	0.1200	2320	5.6324	2320	27.1857	2280
ngcut01$_r$	**243**	0.0000	243	0.0040	243	0.0040	243
ngcut02$_r$	**280**	0.0040	280	0.0000	280	0.0000	280
ngcut03$_r$	**282**	0.0040	282	0.0000	282	0.0000	282
ngcut04$_r$	**416**	0.0040	416	0.0200	416	0.0520	416
ngcut05$_r$	**408**	0.0000	408	0.0160	408	0.0360	408
ngcut06$_r$	**407**	0.0040	407	0.0640	407	0.1840	407
ngcut07$_r$	**1144**	0.0360	1144	0.2160	1144	2.0841	1144
ngcut08$_r$	**1157**	0.0360	1157	0.2720	1157	1.3121	1119
ngcut09$_r$	**1136**	0.0680	1136	0.4280	1136	2.7882	1136
ngcut10$_r$	**2250**	0.0320	2250	0.0000	2250	0.0000	2250
ngcut11$_r$	**2194**	0.2760	2194	1.8281	2194	13.3848	2194
ngcut12$_r$	**2148**	0.5840	2148	1.4881	2148	31.9700	2136
m1$_r$	15054	0.2760	15024	12.1888	15073	85.9094	15024
m2$_r$	**73255**	10.0326	73176	316.3318	73255	6951.4544	73176
m3$_r$	**147386**	0.0760	147386	9.0766	147386	26.5057	142817
m4$_r$	**266233**	0.1160	266233	7.6885	266233	37.6664	265768
m5$_r$	**579883**	0.4600	579883	21.3013	579883	187.6557	577882
gcut01$_r$	**58480**	0.0040	58480	0.0800	58480	0.0880	58136
gcut02$_r$	**61329**	0.0880	61329	12.7728	61329	32.3260	60611
gcut03$_r$	**62020**	0.2840	62020	83.0332	62020	263.8805	61626
gcut04$_r$	**62265**	1.1281	62265	606.8059	62265	2540.2828	62265
gcut05$_r$	**246000**	0.0200	246000	0.5480	246000	1.0081	246000
gcut06$_r$	**243598**	0.0280	243598	1.9801	243598	3.3962	240951
gcut07$_r$	**246988**	0.0840	245912	9.9046	246988	29.9139	245866
gcut08$_r$	**248832**	1.8601	248832	1092.3403	248832	4483.7722	247787
gcut09$_r$	**971100**	0.0080	971100	0.1040	971100	0.1160	971100
gcut10$_r$	**982025**	0.0520	982025	2.8402	982025	4.4403	982025
gcut11$_r$	**984979**	0.4480	984979	102.0784	984979	235.6507	980096
gcut12$_r$	**988694**	1.4161	988694	859.7337	988694	1801.1246	988694
gcut13$_r$	–	–	–	–	–	–	9000000
okp1$_r$	**29080**	790.7494	29080	2457.3536	29080	41298.1610	28876
okp2$_r$	**28794**	8.7485	28794	0.3840	28794	0.3760	28794
okp3$_r$	**29408**	340.3533	29408	1195.3387	29408	43108.8621	29152
okp4$_r$	**485520**	549.7824	485520	1325.5188	485520	89346.3798	477360
okp5$_r$	**28794**	8.5045	28794	0.4040	28794	0.3720	28794
uu1$_r$	**245205**	0.9481	245205	452.1883	245205	1059.5982	242919
uu2$_r$	**595288**	6.6964	595288	2708.2853	595288	19182.6028	595288
uu3$_r$	**1088154**	3.4402	1088154	1594.0036	1088154	7469.8188	1072764
uu4$_r$	**1191071**	42.7667	1188104	34477.7627	1191071	193542.2996	1179050
uu5$_r$	1870038	251.7597	–	–	–	–	1868999
uu6$_r$	**2950760**	10.5407	2950760	2627.2162	2950760	43813.0821	2950760
uu7$_r$	2943852	1785.7636	–	–	–	–	2930654
uu8$_r$	3969784	134.6564	–	–	–	–	3959352
uu9$_r$	6100692	196.0723	–	–	–	–	6100692
uu10$_r$	11995637	914.9932	–	–	–	–	11955852
uu11$_r$	–	–	–	–	–	–	13170382
uw1$_r$	**6916**	1.4681	6916	265.0246	6916	1433.2816	6696
uw2$_r$	**9732**	8.2525	9732	2.4522	9732	2.4842	9732
uw3$_r$	**7188**	4.3923	7188	1.3281	7188	1.4241	7188
uw4$_r$	**8452**	38.6544	8452	8.7005	8452	8.8366	8452
uw5$_r$	**8604**	57.9116	8604	28777.5785	8604	152278.2328	8398
uw6$_r$	**6937**	40.9346	6937	9.1446	6937	9.1126	6937
uw7$_r$	**11585**	68.3443	11585	15.1769	11585	15.1529	11585
uw8$_r$	**8088**	215.8935	8088	37.5984	8088	37.4343	8088
uw9$_r$	7527	228.2463	–	–	–	–	7527
uw10$_r$	8172	409.4536	8172	65.3441	–	–	8172
uw11$_r$	**18500**	5087.5620	–	–	–	–	18200

due to the computational time spent to solve them. Observe that the large CPU time in Tables 3.2 and 3.3 show that some instances are hard even considering the Knapsack problem. Thus, we consider only the instances $cgcut,\ ngcut,\ m$ and $gcut01 - gcut12$. Table 3.4 presents the results for the oriented case of the Cutting Stock problem.

Table 3.4: Results for the 2CSNG problem.

Name	L-approach solution	LB	Difference (%) from LB	Time (s) L-approach	#Columns	Sol_3 solution	Time (s) Sol_3	Improvement (%) over Sol_3	Guillotine solution
cgcut01	37	37.0	0.000%	0.13	57	37	0.10	0.000%	38
cgcut02	44	44.0	0.000%	6.18	79	44	3.43	0.000%	44
cgcut03	365	365.0	0.000%	0.68	106	365	0.11	0.000%	365
ngcut01	35	35.0	0.000%	0.02	25	35	0.02	0.000%	35
ngcut02	101	101.0	0.000%	0.08	23	101	0.02	0.000%	101
ngcut03	111	111.0	0.000%	0.14	84	112	0.12	0.901%	112
ngcut04	42	42.0	0.000%	0.01	13	42	0.01	0.000%	42
ngcut05	82	82.0	0.000%	0.03	25	82	0.02	0.000%	82
ngcut06	96	96.0	0.000%	0.23	106	96	0.18	0.000%	97
ngcut07	33	33.0	0.000%	1.00	39	33	0.54	0.000%	33
ngcut08	60	60.0	0.000%	0.08	57	60	0.04	0.000%	60
ngcut09	146	146.0	0.000%	0.73	80	147	0.22	0.685%	147
ngcut10	72	72.0	0.000%	1.14	17	72	0.30	0.000%	72
ngcut11	36	36.0	0.000%	1.88	69	36	1.76	0.000%	37
ngcut12	121	121.0	0.000%	28.50	109	121	7.80	0.000%	121
m1	97	97.0	0.000%	0.34	66	98	0.27	1.031%	98
m2	86	85.0	1.176%	9.23	166	86	4.46	0.000%	86
m3	94	93.0	1.075%	0.58	80	94	0.38	0.000%	94
m4	71	71.0	0.000%	0.19	51	71	0.11	0.000%	72
m5	101	101.0	0.000%	0.49	65	101	0.20	0.000%	101
gcut01	294	294.0	0.000%	0.03	26	294	0.01	0.000%	294
gcut02	345	345.0	0.000%	0.92	214	345	0.42	0.000%	345
gcut03	332	332.0	0.000%	7.25	528	332	5.67	0.000%	333
gcut04	836	836.0	0.000%	99.29	1790	836	22.17	0.000%	837
gcut05	197	197.0	0.000%	0.10	51	198	0.03	0.508%	198
gcut06	338	338.0	0.000%	0.42	146	339	0.15	0.296%	344
gcut07	591	591.0	0.000%	0.90	159	591	0.36	0.000%	592
gcut08	691	690.0	0.145%	128.49	1547	691	36.35	0.000%	692
gcut09	131	131.0	0.000%	0.12	73	131	0.05	0.000%	132
gcut10	293	293.0	0.000%	0.32	52	293	0.06	0.000%	293
gcut11	330	330.0	0.000%	8.92	582	331	1.93	0.303%	331
gcut12	672	672.0	0.000%	66.53	978	673	19.87	0.149%	672

Observing Table 3.4 (orientated case) we can see that the solution obtained using the L-approach in the column generation coincides with the lower bound (LB) computed, except for the following instances: $m2,\ m3$ and $gcut08$. Moreover, some solutions were obtained in 125 seconds of processing time, in the worst case.

Using the subroutine Sol_3 in the column generation procedure, we could obtain in many cases, solutions equal to those ones computed by the L-approach. It corresponds to 25 out of 32 instances, that is 78.125% of the instances. Moreover, Sol_3 consumed much less CPU time than the use of the L-approach. Note that the CPU time spent, on average, by Sol_3 was 3.35 seconds against 11.40 seconds of the L-approach.

The number of instances in which the L-approach obtained better solutions compared to Sol_3 correspond to 21.875% of the instances. And, the improvement over Sol_3 was of 0.121%, on average, and 1.031% on the best situation for the instance $m1$ (see Table 3.4).

Next, Table 3.5 exhibits the computational results for the problem 2CSNG^r, that is the Cutting Stock problem in which the items can be orthogonally rotated.

Table 3.5: Results for the 2CSNG^r problem.

Name	L-approach solution	LB	Difference (%) from LB	Time (s) L-approach	#Columns	Sol_3 solution	Time (s) Sol_3	Improvement (%) over Sol_3	Guillotine solution
cgcut01$_r$	34	33.0	3.030%	0.24	60	34	0.15	0.000%	34
cgcut02$_r$	44	43.0	2.326%	296.11	184	44	209.15	0.000%	44
cgcut03$_r$	301	300.0	0.333%	67.67	691	301	47.95	0.000%	302
ngcut01$_r$	34	34.0	0.000%	0.02	21	34	0.01	0.000%	34
ngcut02$_r$	98	98.0	0.000%	0.12	48	98	0.08	0.000%	98
ngcut03$_r$	111	110.0	0.000%	0.12	53	111	0.10	0.000%	111
ngcut04$_r$	33	33.0	0.000%	0.06	19	33	0.03	0.000%	33
ngcut05$_r$	65	64.0	1.562%	0.03	31	65	0.03	0.000%	65
ngcut06$_r$	92	92.0	0.000%	0.40	98	93	0.16	1.087%	93
ngcut07$_r$	32	31.0	3.226%	1.77	39	32	1.44	0.000%	32
ngcut08$_r$	56	55.0	1.818%	4.42	88	56	2.37	0.000%	56
ngcut09$_r$	133	133.0	0.000%	5.84	124	134	3.82	0.752%	135
ngcut10$_r$	68	68.0	0.000%	7.99	20	68	1.39	0.000%	68
ngcut11$_r$	35	35.0	0.000%	42.87	80	35	19.54	0.000%	36
ngcut12$_r$	117	116.0	0.862%	66.73	129	117	66.50	0.000%	117
m1$_r$	94	94.0	0.000%	138.80	131	94	36.62	0.000%	95
m2$_r$	83	83.0	0.000%	2030.28	127	83	1968.54	0.000%	84
m3$_r$	84	83.0	1.205%	127.07	196	84	17.76	0.000%	86
m4$_r$	68	67.0	1.493%	19.61	67	68	5.93	0.000%	68
m5$_r$	97	96.0	1.042%	34.69	69	97	23.78	0.000%	97
gcut01$_r$	291	291.0	0.000%	0.18	70	291	0.08	0.000%	291
gcut02$_r$	283	282.0	0.355%	19.87	178	283	9.78	0.000%	283
gcut03$_r$	309	308.0	0.325%	282.75	992	310	267.00	0.322%	315
gcut04$_r$	836	836.0	0.000%	2116.29	1435	836	1301.35	0.000%	836
gcut05$_r$	173	172.0	0.581%	0.97	43	173	0.39	0.000%	175
gcut06$_r$	295	294.0	0.340%	2.92	130	295	2.52	0.000%	302
gcut07$_r$	542	542.0	0.000%	21.05	223	542	10.86	0.000%	544
gcut08$_r$	645	644.0	0.155%	1959.55	1049	645	1459.43	0.000%	651
gcut09$_r$	123	122.0	0.820%	0.34	75	123	0.15	0.000%	123
gcut10$_r$	270	270.0	0.000%	2.70	107	270	2.29	0.000%	270
gcut11$_r$	298	298.0	0.000%	220.80	382	298	95.63	0.000%	299
gcut12$_r$	601	601.0	0.000%	1412.39	597	602	675.55	0.166%	602

Observing Table 3.5 we note that the time spent to solve the instances increased significantly. To compare the differences between the oriented case (see Table 3.4) and this case considering the L-approach, the CPU time was 11.40 and 277.65 seconds, on the average, respectively.

When comparing the lower bound with the solutions returned using the L-approach as subroutine for the column generation, we have a difference of at most 3.226% (for instance $ngcut07_r$). Note that all solutions obtained using the L-approach differ by at most one bin of the lower bound. Moreover, it also returns solutions equal to the lower bound for 16 out of 32 instances. On the other hand, using the subroutine Sol_3, the algorithm returned solutions for 28 instances, that correspond to 87.5%, equal to those ones computed with the L-approach. More details can be seen in Table 3.5.

In order to compare the results presented in Tables 3.4 and 3.5 for the L-approach with those ones computed using algorithm DP2KPG (see column guillotine solution), we obtained better solutions for 15 out of 32 instances of Table 3.4 and 14 out of 32 instances of Table 3.5. However the CPU time spent by L-approach was very high compared to that one required by algorithm DP2KPG, since the algorithm DP2KPG spent no more than 20 seconds against 2120 seconds of the L-approach, in the worst-case.

3.6 Conclusion

In this paper, we presented some recursive approaches for the two-dimensional unconstrained Knapsack and Cutting Stock problems for the more general case in which the cuts are non-guillotine type. We extended some approaches developed for the Manufacturer's Pallet Loading problem: Five-Block heuristic and L-approach.

We also proved the open question raised in the work of Birgin *et al.* (2010) [17], showing that the L-patterns can be generated only using the raster points. We also presented a counterexample in which the L-approach fails to obtain an optimum (non-guillotine) solution. Due to the medium (and large) computational time required by the L-approach to solve some instances, we restricted to a parameter k the depth of its recursive calls and named this variant $L^{(k)}$-approach.

The results obtained for the Two-dimensional Unconstrained Knapsack problem show that the algorithms proposed can improve the guillotine solution considering practical CPU time. Of course, the large instances are hard since the L-approach requires more CPU time than the Five-Block heuristic. Better results were found for the case where the items can be orthogonally rotated.

The highlight was for the Five-Block heuristic that returned equal solutions (for almost all instances) to those obtained by the L-approach, but spending much less CPU time than the latter one. The $L^{(k)}$-approach like the L-approach was not interesting in some instances (large one) because it spend large CPU time and could not improve the previous solution obtained by the Five-Block heuristic. It is also important to say that with these algorithms we improved almost all solutions when compared to the guillotine case.

For the Two-dimensional Cutting Stock problem, the column generation algorithm using the L-approach to generate new columns obtained solutions equal to the lower bound for all instances of the oriented case, except for three instances in which this value differ only one bin. For the case in which orthogonal rotations of the items are allowed, such algorithm obtained solutions that differ only one bin of the lower bound for all instances.

The subroutine Sol_3 used to generate new columns showed to be helpful, because it allowed the column generation approach to obtain in almost all instances the same solution of that using the L-approach, but requiring low CPU time.

The computational results indicate that the algorithms proposed in this paper can be useful

to solve real-world instances of moderate and large sizes. For the instances considered here, the algorithms found optimum or quasi-optimum solutions in a satisfactory amount of computational time.

Future research will be focus on extending the algorithms proposed for the constrained version of Knapsack problem, and also problems like Strip Packing and Cutting Stock with bins of different sizes.

Chapter 4

Exact and Heuristic Algorithms for the Two-dimensional Strip Packing Problem with Order and Static Stability

Abstract

This paper investigates the Two-Dimensional Strip Packing Problem considering the case in which the items should be arranged to form a physically stable packing and also satisfy a predefined order of the items unloading, so that after unloading each item, the packing continues to be stable. We consider the oriented case, where rotations are not allowed. To analyze the packing stability, the presented methodology is based on conditions for static equilibrium of rigid bodies. We formulate an integer linear programming for the Strip Packing problem considering the order constraint, and the stability is dealt by a cutting plane algorithm, leading to a branch-and-cut approach. We also present three heuristics. The first heuristic is based on a stack building algorithm; the second one is based on a branch-and-bound strategy that uses corner points to pack the items; and, the last one is a slight modification of the branch-and-cut approach. The computational experiments show that the exact model is suitable to deal with small and medium-sized instances. With the combination of the heuristics and the branch-and-cut algorithm some instances were solved in few seconds. Optimal solutions were obtained after few minutes (and hours) of CPU processing.

4.1 Introduction

Packing problems have many applications in the industry and real-life problems, and in many cases, they include a large set of constraints. In [18], Bischoff and Ratcliff (1995) outlines some practical considerations for packing problems. One of them is the load stability, also referred to as cargo stability constraint (see [58]). The cargo stability aims to pack items in which any item has its bottom face supported by the top faces of other items or the bottom of the bin, so that no item can rotate or lay down after it is packed.

In other words, the cargo stability constraint involves the packing of items into a bin in a physically stable way that satisfy the static equilibrium conditions of rigid bodies. In a more realistic scenarios, the dynamic equilibrium is considered instead of the static one. From now on, when we refer to static (cargo) stability or static equilibrium we are considering the static equilibrium of rigid bodies.

Another common constraint involves an *order (or sequence)* in which the items must be loaded/unloaded from the bin. In fact, for some industries, the unloading of boxes from trucks or containers is performed by machines and only occurs from one side (lateral-side or up-side). As a result, the boxes that must be unloaded first have an order (number) higher than the last ones. From now on we refer to this constraint as *order constraint*. Applications involving the order constraint can be found in some variants of the Vehicle Loading problems [40, 55]

This paper investigates the Two-dimensional Strip Packing problem subject to the order and static stability constraints.

We assume for the order constraint that the items will be (un)loaded from the up-side of the strip. In this case, an item i with order o_i must be placed at the bottom of the strip or over items whose order value are smaller than or equal to o_i. Then, for any item i there must exist free space to remove the item in the direction of the unloading side, and no item j with $o_j > o_i$, is packed above i obstructing it.

For the static stability constraint, we assume that each item i has also a value of mass m_i. To ensure the stability of the items the concepts about strength of materials, that is the formulations used to analyze beams under the static equilibrium of rigid bodies, are considered [12, 82].

TWO-DIMENSIONAL STRIP PACKING PROBLEM WITH ORDER AND STATIC STABILITY CONSTRAINTS (2SPOS): Given a strip $B = (L, \infty)$ and a set of rectangles S, each rectangle i with dimensions (l_i, w_i), order value o_i and mass m_i, find a packing of S into the strip B so that the order and static stability constraints are satisfied and the height of the used portion of the strip is minimized.

The formal definition of static stability are presented in the Section 4.2.

In a feasible packing the items cannot overlap and they must be packed in an orthogonal way. That is, the sides of the items must be parallel or orthogonal to the sides of the strip. Moreover, the items cannot be rotated, and the order and static stability constraints must be satisfied. The

packings are considered in the Cartesian plane $\mathbb{R}^2$, and a rectangle is specified by a pair (x, y), where x and y denote, respectively, the length and height of any object. The position $(0, 0)$ represents the bottom left corner of the strip B, and the position of an item is specified by its bottom left corner. We also assume that the values in each instance are all integer numbers.

The Two-dimensional Strip Packing problem, denoted by 2SP, is NP-hard as shown in [37], and it is a particular case of the 2SPOS problem when $o_i = 0$, for $i = 1, \ldots, n$ and the constraint about the static stability is not considered. Following the typology of Wäscher *et al.* [86], this is the *Two-dimensional Rectangular Open Dimension Problem*.

Some algorithms have been proposed for the 2SP problem. Kenyon and Rémila (2000) [61] presented an AFPTAS for the oriented case and Jansen and van Stee (2005) [57] proposed a PTAS for the case in which rotations are allowed. Hifi (1998) [50], Lodi *et al.* (2003) [66], Martello *et al.* (2003) [67] and Lesh *et al.* (2004) [64] presented approaches using branch-and-bound and integer linear programming models. In the work of Lesh *et al.* (2004) [64], the main idea of the proposed branch-and-bound algorithm is to prune branches when it is impossible to pack a new object without generating a hole. Cintra *et al.* (2008) [27] presented a column generation based algorithm for the staged 2SP problem with and without rotations.

Recent surveys for the 2SP problem can be found in Riff *et al.* (2009) and Ntene and Vuuren (2009) [81, 75]. In 2009, Kenmochi *et al.* [60] proposed an exact branch-and-bound algorithm for the oriented case and the case in which the items can be rotated orthogonally. For the oriented case, the proposed algorithm is faster than the one presented in [64]. Ortmann *et al.* (2010) [77] presented four new and improved level-packing algorithms.

Most results presented for the Strip packing problem consider heuristic approaches. The main reason may be due to the restricted size of the instances solved by exact algorithms. Although heuristics can solve large-sized instances, in most of the cases they cannot guarantee any information about the optimality of the obtained solution. When considering metaheuristics, there are genetic algorithms, simulated annealing methods and GRASP [3, 51, 53, 87].

We found few papers related to the cargo stability in the literature. Junqueira *et al.* (2010) [58] presented an integer linear formulation for the problem of loading rectangular boxes inside bins under cargo stability and load bearing constraints. They define a cargo stability constraint based on a factor α of bottom contact. This factor measure the percentage of the bottom face of each item that must be in contact with other items or the strip. This same requirement is used by Gendrau *et al.* (2006) [39] when dealing with the three-dimensional Loading Capacitated Vehicle Routing problem. Naturally, the above condition is an approximative model and may obtain packings that do not satisfy the conditions for the static equilibrium of rigid bodies.

On the other hand, we only found the work of Castro Silva *et al.* (2003) [85] for the packing problem with cargo stability constraints that consider the static equilibrium of rigid bodies, as considered in this paper. The authors proposed a greedy heuristic for the three-dimensional Bin Packing problem with the constraint that the items packed within the bins has to be physically

stable. The heuristic is based on three ideas: looks for a best filling of a single bin by the items; two items cannot overlap; and an item is inserted into a bin in a (feasible) position which generates the minimum increase on the unused space of the overall packing volume. To insert a new item a subroutine is called to verify if the item remains stable as well as the packing. They tried to incorporate the dynamical stability, without any success as mentioned by themselves.

We propose a slightly different approach from the one proposed by Castro Silva *et al.* (2003) about the static stability. Our methodology allows not only to verify the static stability, but also to compute the forces (any kind of force acting on the items/packing) that a single item exerts to the other ones under it. As a result, our methodology may be extended to deal with the dynamical stability, namely, the cases in which the items or packing have variations of velocity, acceleration, inclinations and/or curvatures.

In this paper, we propose three heuristics and a branch-and-cut algorithm for the 2SPOS problem. The first heuristic consider the packing of items into vertical layers. The second one is derived from the exact branch-and-bound algorithm proposed by [68], and the last one is a slight modification of the branch-and-cut algorithm proposed here. The heuristics are discussed in Section 4.5.

Due to the complexity of the static stability constraints, we propose an integer linear formulation for the 2SPOS problem, that initially starts without any static stability constraint. To avoid integer solutions that corresponds to unstable packings the algorithm add constraints (cutting planes) into the initial program. More details are presented in Section 4.3.

The computational experiments are given in Section 4.6. We report numerical experiments for some random instances and other instances presented in the OR-Library [10]. The results obtained confirm the good performance of the proposed algorithms. Finally, Section 4.7 is devoted to the conclusions and suggestions for future works.

4.2 Static Stability in Packing Problems

To deal with the static stability of the items/packing, we use some concepts about strength of materials that include static equilibrium of rigid bodies. So, our methodology is derived from the physical concepts of force, center of gravity, moment, bending moment, beams and the three-moment equation method [21, 47, 49].

Without loss of generality, the rectangular items are associated with structural elements called *beams*. The aim is to use the concepts and formulations of such elements to deal with the static stability in packing problems. So, we do not make any difference between the terms (rectangular) items and (simple continuous) beams. Moreover, the proposed methodology requires some assumptions that can be easily used/extended to any packing problem.

First, we only consider the existence of the weight force, produced by each item. That is, we only consider the weight of each item for the static stability. In others scenarios there may exist

others kind of forces, for example the wind action, or in cases of transportation, the dynamical action of the truck over the packing, etc.

The weight of an item corresponds to the gravity force acting on that item and it is uniformly distributed along the item. This force can be substituted by only one resulting force that acts on the center of gravity of the item. As we consider the static equilibrium of rigid bodies the forces do not vary with the time (are static) and the items are rigid (rectangular) bodies.

The mass m of the items can be assumed to be concentrated in only one point that is called *center of mass*. Under the consideration that the gravitational field is uniform, the center of gravity coincides with the center of mass. Then, the center of mass is the point where the resulting weight force acts on. The Eq. (4.1) is used to calculate the center of mass of an item composed by other masses:

$$\overrightarrow{CM} = \frac{\sum_{i=1}^{n} \overrightarrow{r_i} m_i}{\sum_{i=1}^{n} m_i} \quad \Rightarrow \quad \begin{array}{l} CM_x = \frac{\sum_{i=1}^{n} r_i^x m_i}{\sum_{i=1}^{n} m_i} \\ \\ CM_y = \frac{\sum_{i=1}^{n} r_i^y m_i}{\sum_{i=1}^{n} m_i} \end{array} \tag{4.1}$$

where $\overrightarrow{CM}$, with components in the x- and y-axes, is the vector that indicates the center of mass of the item; m_i is the ith mass and whose distance to an inertial reference frame (First Newton's Law) is described by the position vector $\overrightarrow{r_i} = (r_i^x,\ r_i^y)$. The bottom left corner of the item is adopted as its inertial frame.

Any item without other forces acting on it has its center of mass located at the mid-point of its dimensions. We say that an item i is adjacent to an item j if j is immediately under and in direct contact with i. From now on, an item composed by one mass is in static equilibrium (for the sake of simplicity, is stable) when its center of mass lies on one of the following *stable regions*: over some of its adjacent items; in the middle of two (or more) of its adjacent items; or, in the floor of the bin. If the center of mass of an item does not lie on any of these stable regions a movement (of the item) occurs, and consequently the item may fall down. Note that the lateral borders of the strip are not considered, so that they do not restrict any item to rotate, and fall down.

The modulus of the weight force for an item i with dimensions (l_i, w_i) is calculated by Eq. (4.2), since the items are homogeneous and made of the same material.

$$F_{p_i} = m_i g \quad (N) \tag{4.2}$$

where g is the acceleration of gravity (e.g., $g = 9,82\ m/s^2$). We also assume that the mass of an item i is given by its area. More precisely, for each item i, we have $m_i = l_i c_i$.

To verify the static stability of any packing, it is necessary to analyze each item individually verifying if the center of mass lies on a stable region. Moreover, if an item is stable, the resultant

force acting on its center of mass is then carried to its adjacent items. On the other hand, if an item is unstable the packing is assumed to be unstable, since such item will rotate and may break up the packing.

To determine the value of the forces transmitted from the item under consideration to its adjacent ones the equations for static equilibrium of rigid bodies are applied. An item is in static equilibrium when the resultant of the forces and moments acting on it are equal to zero:

$$\sum_{i=0}^{n} \overrightarrow{F_i} = \overrightarrow{0}; \qquad \sum_{i=0}^{n} \overrightarrow{M_i^O} = \overrightarrow{0}, \tag{4.3}$$

where $\sum_{i=0}^{n} \overrightarrow{F_i}$ and $\sum_{i=0}^{n} \overrightarrow{M_i^O}$ represent the vectorial sum of all the forces and moments acting on item i, respectively. The bottom left corner of the item under consideration is adopted as the inertial frame O whenever the moment is computed. As we use the (2D) Cartesian plane and only the weight forces (whose direction is vertically downward) are acting on any item, only the y direction (of the force) and z direction (of the moment) need to be considered.

Therefore, the objective is to determine how to transmit the resultant forces of the item under consideration to its adjacent ones, and how to determine the place where such forces will act on them. As a consequence, there are three cases to analyze:

- (i) the item has exactly one adjacent item;
- (ii) the item has exactly two adjacent items; and,
- (iii) the item has three or more adjacent items.

The cases (i) and (ii) appeared frequently on numerical experiments, and they are basic cases for the last one. In particular, the cases (i) and (ii) have straightforward formulations that are helpful and allows the subroutine to check the stability constraints very quickly.

The *hypothesis of small displacements* is assumed, since beams are structural elements in which forces and loads act on, they may deform and/or move from its original position. Therefore, it means that the conditions for static equilibrium are imposed on the original geometry (undeformed) of the structure [12, 21].

4.2.1 Case (i)

The case (i) is the most simple case and consists in one or more items over exactly one adjacent item. Let i be an item in which only its weight force $\overrightarrow{F_{p_i}}$ is acting on its center of mass $\overrightarrow{CM_i}$, and let j be the unique adjacent item of i. Assume that the center of mass of item i is positioned on a stable region (that is i is stable). This situation is represented in Fig. 4.1.

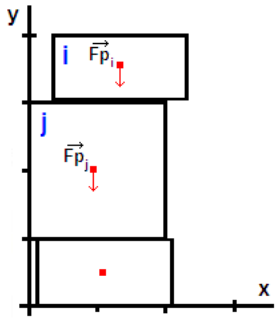

Figure 4.1: Simple representation of case (i).

Now, to analyze item j it is necessary to consider not only its weight force acting on its center of mass, but also the forces that are acting from the items immediately over (and in direct contact with) item j. In this example, there is only the weight force of the item i. In this case, the point where the force of item i is acting on item j corresponds to the point where the center of mass of i is located.

The item j has more than one force acting on, and then, its center of mass must be calculated using Eq. (4.1). If such center of mass lies on a stable region, then item j is also stable and its resultant force can be carried to its adjacent items (of course, following some of the three cases above mentioned). On the other hand, item j is unstable and the algorithm that verify the stability can stop.

4.2.2 Case (ii)

The case (ii) configures the situation in which an item has exactly two other adjacent items. In this situation the analogy with beams simplify the analysis and the configuration can be verified directly. Let k be the item to be analyzed where three forces are acting on: its own weight force on its center of mass $\overrightarrow{CM_k}$, and two others forces: $\overrightarrow{F_{p_i}}$ and $\overrightarrow{F_{p_j}}$ applied on points $\overrightarrow{p_i}$ and $\overrightarrow{p_j}$, respectively. Consider that items k_1 and k_2 are under item k. Figure 4.2 represents this case using the rectangular items and also the analogy with beams.

Let us assume that the center of mass of item k lies on a stable region. Now, we have to calculate the resultant forces of k to its adjacent items.

As shown in Fig. 4.2 the item k represents a simple continuous beam and its adjacent items, k_1 and k_2, are its supports (for example, pillars) that interacts with it. In such interaction two new forces appear, R_1 and R_2, so called *reaction forces*.

From now on, we refer to *the resultant force* as the resultant of all forces that acts on the beam, except the reaction forces. Then, to satisfy the Newton's third law and, consequently, the conditions for static equilibrium the resultant force must be decomposed into forces acting on the adjacent supports on contact with the beam. Without loss of generality, we assume that the

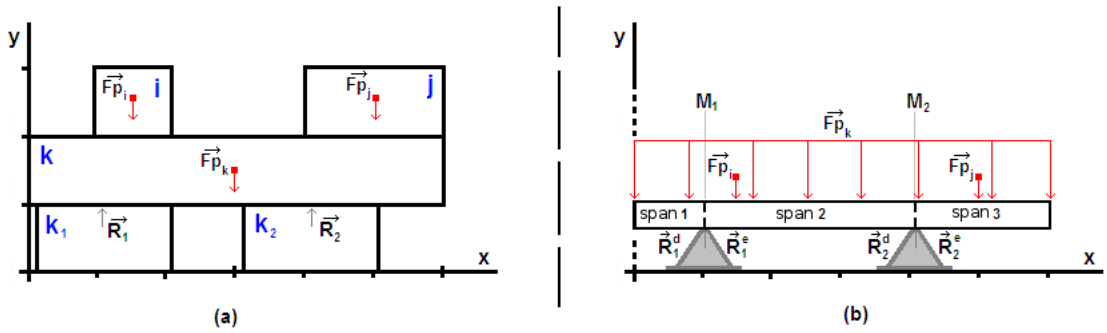

Figure 4.2: (a) Example of the case (ii) using rectangular items; and, (b) its representation by a simple continuous beam.

point where each "partial" resultant force (obtained after the decomposition) acts on its support corresponds to the mid-point of the length of contact between the beam and the support under consideration, as presented in Fig. 4.2(a).

In order to calculate the forces carried from the item under consideration to its adjacent ones, namely, to calculate the reaction forces R_1 and R_2, the analysis consists to divide the beam into three parts, so called *spans (segments)*: span 1 (to the left of the support R_1); span 2 (between supports R_1 and R_2); and, span 3 (to the right of the support R_2).

The equations for static equilibrium, Eq. (4.3), are first applied to the spans in the extremes (in the present situation they are the spans 1 and 3). Therefore, the moments and the first part of the reaction forces: R_1^d and R_2^e, are computed. Next, the Eq. (4.3) is applied on the intermediary spans (only span 2 in this case), and the last part of the reaction forces: R_1^e and R_2^d, can be computed. Finally, using the *superposition principle* [47], the final value of the reaction forces of the supports can be obtained. And, they correspond to the forces carried from the item under consideration to its adjacent ones.

4.2.3 Case (iii)

This last case is a natural extension of the case (ii). The main difference appears on the number of adjacent items. In this situation the item under consideration has three (or more) adjacent items, and then, the simple continuous beam has three (or more) supports that characterizes a *hyperstatic beam*. For this beam, the number of unknown variables (reaction forces) to be computed is more than the number of equations for static equilibrium available.

The Eq. (4.3) allows to solve only the cases (i) and (ii) in a direct way, because there are two equations. For the case (iii) it is necessary to compute the (internal) moments that appear on the intermediate supports. If the moment in these supports are known, then it is possible to divide the beam into span segments, and thus apply the equations for static equilibrium on each

segment.

We can apply a simple method called *Three-Moment Equation method* used to deal with hyperstatic structures, and then, calculate the reaction forces on the supports [21, 38].

To deal with the static stability, a beam with n span segments has exactly $n-1$ supports, since the spans in the extremes (spans 1 and n) are free of supports. The spans in the extremes are not considered when applying the three-moment equation method, because the moments M_1 and M_{n-1} on respective supports 1 and $n-1$ are calculated directly by Eq. (4.3). Furthermore, the first part of the reaction forces R_1^d and R_{n-1}^e are also computed directly.

The three-moment equation method is applied in adjacent pair of spans (i.e., two by two subsequent span segments) starting from the span 2 to the span $n-1$. Figure 4.3 schematically represents an item k with three adjacent items, k_1, k_2 and k_3. Note that there are four spans and three supports.

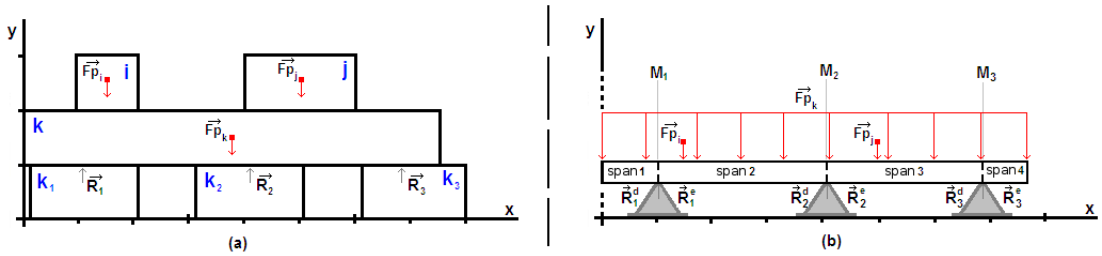

Figure 4.3: Example of the case (iii) for an item with others three adjacent items.

In what follows we describe the matrix formulation of the three-moment equation method for the case in which the beam has n span segments over $n-1$ supports. This situation is represented by an item with $n-1$ adjacent items. Note that individual forces are acting on each span of the beam as its own weight force and/or the resultant forces that goes from the items immediately above. Then, it is only necessary to solve the following equation symbolically:

$$\mathbf{F}\,\mathbf{m} = \mathbf{d}, \tag{4.4}$$

where $\mathbf{F}$ is the *flexibility matrix*. The matrix F is three-diagonal and symmetric; $\mathbf{m}$ is the vector of (internal) moments at the supports; and, $\mathbf{d}$ is the vector of forces/loads applied on the item under analysis. The objective is to compute the system of equations (4.4) to find the values in vector $\mathbf{m}$.

Starting from the span 2 to the span $n-1$ and considering that support j is between span j and span $j+1$, the three non-zero elements in row j ($j = 2, \ldots, n-2$) of matrix F are given by:

$$\begin{aligned} F_{j,j-1} &= l_j, \\ F_{j,j} &= 2(l_j + l_{j+1}), \\ F_{j,j+1} &= l_{j+1}, \end{aligned} \tag{4.5}$$

where l_j is the length of the span j. The jth position of the vector d corresponds to:

$$d_j = -6(\mu_j^2 + \mu_{j+1}^1), \tag{4.6}$$

where μ_j^2 and μ_{j+1}^1, are respectively the (static) load factors of the span j for the right support (entry 2 in the exponent), and of the span $(j+1)$ for the left support (entry 1 in the exponent).

The load factors are calculated from the forces (except the reaction forces) acting on each span segment. There are some cases in which two or more forces are acting on the span. For instance, when we have the weight force of a span and the forces of the other items that are over and in direct contact with the span. So, the final load factor is computed by summing up all load factors acting on the item. Figure 4.4 presents the two ways to calculate the load factors: (a) for an uniformly distributed force/load, e.g., the weight force; (b) for a concentrated force/load, e.g., the resultant forces of the items over and in direct contact with.

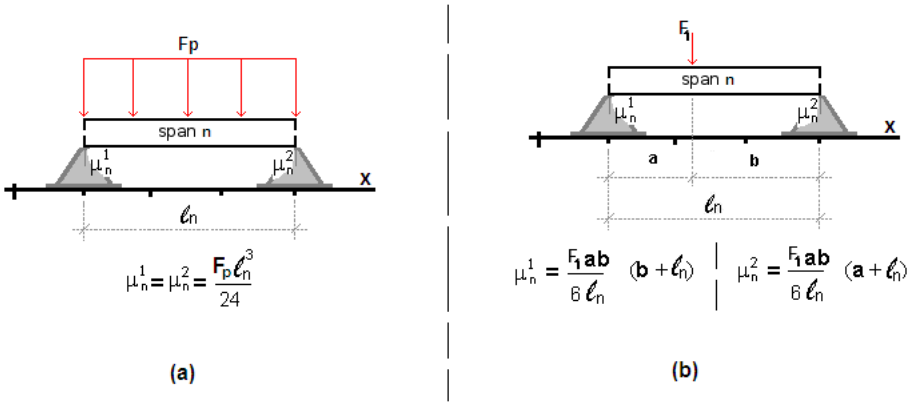

Figure 4.4: Equations to compute the load factors: (a) for an uniformly distributed force/load; (b) for a concentrated force/load.

The following values $F_{1,1}$, $F_{n-1,n-1}$, d_1 and d_{n-1} are given by:

$$\begin{aligned} F_{1,1} &= 1 \\ F_{n-1,n-1} &= 1 \\ d_1 &= -(F_p^1 \frac{l_1}{2}) - \sum_{i=1}^{r_1} F_i^1 d_i^1 \\ d_{n-1} &= -(F_p^{n-1} \frac{l_{n-1}}{2}) - \sum_{i=1}^{r_{n-1}} F_i^{n-1} d_i^{n-1}, \end{aligned} \tag{4.7}$$

where F_p^1 and l_1 represent, respectively, the weight force and the length of the span 1; r_1 is the number of forces carried from the items over and in direct contact with the span 1 (in this case the weight of the span is not considered); and, F_i^1 and d_i^1 are, respectively, the modulus of the ith force and its distance from the inertial frame, both for the span 1. The same remains for F_p^{n-1}, l_{n-1}, r_{n-1}, F_i^{n-1} and d_i^{n-1} considering the span $n-1$.

Using the values of F and d, the moments **m** can be obtained using Eq. (4.4). Applying the equations for static equilibrium for each span segment, we can compute the reaction forces from adjacent items.

4.2.4 The Algorithm

Considering the three cases above mentioned, we have all information to determine the forces carried for each item, and then, to verify if a two-dimensional packing is statically stable. The Algorithm 4.1 described in the following allow us to verify the stability.

The worst-case time complexity of the Algorithm 4.1 is $O(n^4)$ where n is the quantity of items in the input packing I. The steps of the lines $4.1.1 - 4.1.8$ can be executed in time $O(n \log n)$ considering any $O(n \log n)$ algorithm to sort the items. The loop in lines $4.1.9 - 4.1.26$ spend time $O(n^4)$. Note that for case (iii) the algorithm obtain the inverse of the flexibility matrix **F**, which can be computed in time complexity $O(n^3)$.

Algorithm 4.1: Algorithm to verify the static stability of a 2D packing.

Input : Packing I of a set of n rectangular items.
Output : It returns if I is (or not) statically stable.

4.1.1 Let $p_i(x_i, y_i)$ be the left bottom corner for each $i \in I$.
4.1.2 Let $S(I)$ be the arrangement of the items in decreasing order of y_i. Items with the same height are arranged in increasing way by x_i.
4.1.3 Let $A(i)$ be the set of items that are adjacent to i, for each $i \in S(I)$.
4.1.4 **foreach** $i \in S(I)$ **do**
4.1.5 **foreach** $j \in A(i)$ **do**
4.1.6 Let I_{ij} be the segment that represents the length of contact between items i and j.
4.1.7 Let (x^s_{ij}, y^s_{ij}) and (x^e_{ij}, y^e_{ij}) be the initial- and end-points of I_{ij}, respectively.
4.1.8 Let $(x^p_{ij}, y^p_{ij}) = (\frac{(x^e_{ij} - x^s_{ij})}{2}, \frac{(y^e_{ij} - y^s_{ij})}{2})$ be the mid-point of I_{ij}.
4.1.9 **foreach** $i \in S(I)$ **do**
4.1.10 **if** *there are no forces carried from items over and in direct contact with item* i **then**
4.1.11 Compute the center of mass $\overrightarrow{CM}_i$ of the item i.
4.1.12 **if** $\overrightarrow{CM}_i$ *does not fit on a stable region* **then**
4.1.13 **return** (Item i is unstable; the packing is unstable).
4.1.14 **else**
4.1.15 Compute the center of mass $\overrightarrow{CM}_i$ of the item i by Eq. (4.1), considering the set of forces carried from items over and in direct contact with item i.
4.1.16 **if** $\overrightarrow{CM}_i$ *does not fit on a stable region* **then**
4.1.17 **return** (Item i is unstable; the packing is unstable).
4.1.18 Let $k \leftarrow |A(i)|$ be the quantity of adjacent items to item i.
4.1.19 Compute (and save) the forces carried from item i to its adjacent items that belong to $A(i)$ for the exact value of k, that is:
4.1.20 **if** $k = 1$ **then**
4.1.21 Apply **case (i)**.
4.1.22 **else**
4.1.23 **if** $k = 2$ **then**
4.1.24 Apply **case (ii)**.
4.1.25 **else**
4.1.26 Apply **case (iii)**.
4.1.27 **return** The packing is stable.

4.3 2SPOS **Problem**

In what follows we present the integer linear formulation and heuristics for the 2SPOS problem. In this formulation, the strip $B = (L, \infty)$ is discretized in a grid of points (non-necessarily uniform).

The discretization of the strip $B = (L, \infty)$ generates a grid of points that we denote by $\mathcal{P}$. All the horizontal lines (in the height direction, y-axis) are in the set $\mathcal{W}$ and all the vertical lines

(in the length direction, x-axis) are in $\mathcal{L}$. In the x-axis, we may assume a uniform discretization, whose distance between subsequent points is given by a sufficiently small value d_x. For the y-axis, we may assume the set $\mathcal{W}$ as the set of all combinations between the height of the items. The coordinates of a point $p \in \mathcal{P}$ are given by a pair $p = (a, b)$ where $a \in \mathcal{L}$ refers to the x-axis, and $b \in \mathcal{W}$ refer to the y-axis.

As a consequence, the problem consists find an optimal packing of the items within the grid of points in such a way to minimize the overall height of the strip, satisfying the order and the static stability constraints.

4.3.1 The Integer Linear Formulation

Now, we present an integer linear programming model, described in Eq. (4.8), that formulates the 2SPOS problem when each item is packed in a discretization point. An instance of the 2SPOS problem is denoted by $I = (L, \infty, l_{1,\dots,n}, w_{1,\dots,n}, o_{1,\dots,n})$, where each item i has dimensions (l_i, w_i) and value of order $o_i > 0$, for $i = 1, \dots, n$. Let $\mathcal{P}_i \subseteq \mathcal{P}$ be the set of feasible points where the item i can be packed without passing the border of the strip. The binary variable z_e is equal to 1 if a point $p_e \in \mathcal{P}$ for some horizontal line $e \in \mathcal{W}$ is covered by some item, and $z_e = 0$, otherwise. The binary variable x_{ip} is 1 if the item i is packed with its bottom left corner on point $p \in \mathcal{P}$.

We denote by $\mathcal{R}_{ip} \subseteq \mathcal{P}$ the set of points of the grid that are covered by the item i, when it is packed in point p, except those ones that matches the upper and right border of i. The binary variable y_{qi} indicates whether the point $q \in \mathcal{P}$ is covered by item i.

The set $\mathcal{O} = \{1, \dots, \mathcal{O}_{\max}\}$ contains possible order values for each item and value $o_i \in \mathcal{O}$ is the order of item i. The binary variable r_{po} indicates whether point $p \in \mathcal{P}$ has value of order $o \in \mathcal{O}$. Note that if $x_{ip} = 1$, then all the points $q \in \mathcal{R}_{ip}$ have value of order equal to o_i (of item i). For each point $p = (a, b) \in \mathcal{P}$, we denote by $\lambda(p)$ the point $p' = (a, b')$ where $b' = \min\{\beta \in \mathcal{W} \mid \beta > b\}$.

It is desirable that each item is packed over another item, or in the bottom of the strip, in such a way to avoid items "floating in mid-air" within the packing. Naturally, items floating in mid-air generate non-stable packings.

Therefore, for an item i with dimensions (l_i, w_i) packed on point $p = (a, b)$, let $\mathcal{S}_{ijp}$ be the set of points in which the item j with dimensions (l_j, w_j) can be packed (these points are under item i) to ensure that item i always have an extension of contact with j of at least d_x units. In other words, the bottom border of the item i must have a nonempty extension of contact with the upper border of item j. The points in $\mathcal{S}_{ijp} \subseteq \mathcal{P}$ are the points with abscissa in $(a - l_j,\ a + l_i)$, with $0 \leq a - l_j \leq L$ and $a - l_j \leq a + l_i \leq L$.

The objective function in formulation (4.8) is composed by two parts. The first one aims to minimize the overall height of the strip. The second complements the first one in the sense of

keeping the items always close to the bottom border of the strip (floor). The value of ϵ is a very small number.

$$\min \quad \sum_{e=1}^{|\mathcal{W}|} e z_e + \epsilon \Big(\sum_{i=1}^{n} \sum_{p=(\bullet,b)\in\mathcal{P}_i} b x_{ip}\Big)$$

subject to :

$$(1) \quad \sum_{p\in\mathcal{P}_i} x_{ip} = 1 \qquad \forall\, item\ i\ (i = 1,\ \ldots,\ n).$$

$$(2) \quad \sum_{i=1}^{n} y_{pi} \leq 1 \qquad \forall\, p \in \mathcal{P}.$$

$$(3) \quad \sum_{o=1}^{\mathcal{O}_{\max}} r_{po} \leq 1 \qquad \forall\, p \in \mathcal{P}.$$

$$(4) \quad x_{ip} \leq r_{qo_i} \qquad \forall\, item\ i\ (i = 1,\ \ldots,\ n);\ \forall\, p \in \mathcal{P}_i;\ \forall\, q \in \mathcal{R}_{ip}.$$

$$(5) \quad x_{ip} \leq y_{qi} \qquad \forall\, item\ i\ (i = 1,\ \ldots,\ n);\ \forall\, p \in \mathcal{P}_i;\ \forall\, q \in \mathcal{R}_{ip}.$$

$$(6) \quad r_{p'o'} \leq \sum_{o''=o'}^{\mathcal{O}_{\max}} r_{p''o''} \qquad \forall\, p' \in \mathcal{P};\ \forall\, o',\ o'' \in \mathcal{O};\ such\ that\ o' \leq o'',\ p'' = \lambda(p'). \tag{4.8}$$

$$(7) \quad x_{ip} \leq z_e \qquad \forall\, item\ i\ (i = 1,\ \ldots,\ n);\ \forall\, p \in \mathcal{P}_i;\ \forall\, e \in \mathcal{N}_{ip}.$$

$$(8) \quad x_{ip} \leq \sum_{\substack{j=1 \\ j\neq i}}^{n} \sum_{q\in\mathcal{S}_{ijp}} x_{jq} \qquad \forall\, item\ i\ (i = 1,\ \ldots,\ n);\ \forall\, p \in \mathcal{P}_i.$$

$$(9) \quad \sum_{ip\in P} x_{ip} \leq n - 1 \qquad \forall \text{ non-stable packing } P,$$

$$(10) \quad x_{ip} \in \{0, 1\} \qquad \forall\, item\ i\ (i = 1,\ \ldots,\ n);\ \forall\, p \in \mathcal{P}.$$

$$(11) \quad y_{qi} \in \{0, 1\} \qquad \forall\, item\ i\ (i = 1,\ \ldots,\ n);\ \forall\, q \in \mathcal{P}.$$

$$(12) \quad r_{po} \in \{0, 1\} \qquad \forall\, o \in \mathcal{O};\ \forall\, p \in \mathcal{P}.$$

$$(13) \quad z_e \in \{0, 1\} \qquad e = 1,\ \ldots,\ |\mathcal{W}|.$$

Constraints $(4.8.1)$ impose that each item i is packed exactly once. Constraints $(4.8.2)$ ensure that each point of the grid is covered by at most one item, and constraints $(4.8.3)$ impose

that such points have at most one value of order. It is easy to see that there may exists points of the grid that will not be covered by any item.

When a item i is packed in the point p, constraints $(4.8.4)$ impose that all the points in $\mathcal{R}_{ip}$ have the same value of order o_i, and constraints $(4.8.5)$ ensure that such points are marked in such a way that no item can be packed on them except item i.

To ensure the order constraint imposed by the problem, constraints $(4.8.6)$ impose that any item i with order o_i must be packed over items j with order o_j that satisfy $o_j \leq o_i$. In other words, if a point $p = (a, b)$ has value of order o_i, then the point immediately over it, say point $q = (a, c)$ (on the same line that intercepts at y-axis), must have value of order o_j, with $o_j \leq o_i$.

Constraints $(4.8.7)$ ensure that the horizontal lines of the grid covered by any item has to be used. Observe that the item $i = (l_i, w_i)$ packed on point $p = (a, b)$ covers the horizontal lines in $\mathcal{N}_{ip} = [b,\ b + \lceil \frac{w_i}{d_y} \rceil]$.

To avoid items floating in mid-air, constraints $(4.8.8)$ ensure that an item i will be packed only if there exist an extension of contact with other item j or if item i lies on the floor of the strip. Constraints $(4.8.9)$ avoid packings that are not stable. Finally, constraints $(4.8.10-4.8.13)$ ensure that all variables are binary.

4.4 The Branch-and-Cut Algorithm

Since the number of non-stable packings is very large, we do not insert all the static stability constraints in the integer formulation (4.8). As we must obtain packings for the 2SPOS problem that are stable, the static stability constraints are inserted on demand as cutting planes. For the sake of simplicity, we call this algorithm as BCut.

The integer formulation used in the algorithm BCut is solved by the commercial software CPLEX. However, any integer linear programming solver that can deal with cutting planes can be used. In fact, most of the commercial software also provide a framework for branch-and-cut algorithms. In this framework, we implemented a cutting plane routine that is called every time a new (integer) solution candidate is obtained. The routine first verify, using Algorithm 4.1, if the corresponding packing is stable. In case positive, the new solution is accepted. Otherwise, such candidate solution is avoided inserting the following inequality in the linear program:

$$\sum_{i=1}^{n} x_{ip} \leq n - 1, \tag{4.9}$$

where item i was packed on point p in such solution. The cuts of stability are inserted as global cuts in the branching tree, so that they are valid for all packings.

Besides the cuts of stability, the CPLEX solver automatically manage and insert other types of cuts. We call these cuts “standard cuts”. Among the standard cuts, the solver may insert:

clique cuts, cover cuts, disjunctive cuts, flow cuts, Gomory cuts and zero-half cuts.

4.5 The Heuristics

Naturally, the number of variables and constraints in formulation (4.8) depends on the discretization of the strip. To avoid unnecessary computation we present some heuristics to compute the initial height for the grid as well as feed the solver with an initial solution (upper bound) for the integer formulation.

We denote the first heuristic by HffO, and is defined as follows: Given an instance $I = (L, \infty, l, w, o)$ for the 2SPOS problem, the algorithm HffO considers the items sorted decreasingly by length $(l_1 \geq l_2 \geq \ldots \geq l_n)$. For the sake of simplicity, we present this algorithm in two phases. The first phase consists in packing the items on the floor of the strip. As the items were sorted decreasingly by length, the first item with length l_1 will be packed on the point $(0, 0)$, the second item on the point $(l_1, 0)$, the third item on the point $(l_1 + l_2, 0)$, and so on. When the packing of the next item of the instance extrapolates the length of the strip, it is ignored and the routine continues from the next item.

The first phase aims to create vertical layers whose length corresponds to the length of the items already packed there. Suppose we have k items packed on the floor, with lengths $l_{i_1}, l_{i_2}, \ldots, l_{i_k}$. Each one of these items will start a vertical layer of items, where layer j has length l_{i_j} and height given by the sum of the heights of the items in the layer. Initially, layer j has height w_{i_j} given by the unique item i_j.

In the second phase, the algorithm sorts the non-packed items decreasingly by height. To pack the next item, the algorithm looks for a layer with sufficient length and smallest height. Ties are broken by choosing the layer with smallest possible length. The item is packed in the middle of the layer length.

Algorithm HffO always generates stable packings, since each item is packed in the middle of the layer. Now, to generate packings that also satisfy the order constraints, the algorithm sort the items in the layer decreasingly by the value of order. The Algorithm 4.2 describes the heuristic HffO.

The heuristic HffO can be improved for the case in which there is no order value (or all items have a same order). For each item to pack, a new vertical layer is created. Observe that the items were sorted decreasingly by length, so that to pack the item i, whose length is l_i, in layer k, whose length is d_k, we have $l_i \leq d_k$. If $l_n \geq d_k - l_i$, then a new vertical layer of length $d_k - l_i$ can be created, since at least one item (the last one) can be packed in it. This is the only modification to be made in algorithm HffO, so that the other steps remain, except those ones about the order constraint (lines $4.2.18 - 4.2.22$) that must be deleted. This version of the algorithm is denominated by HffNo.

It is clear that algorithms HffO and HffNo has worst-case time complexity equal to $O(nlog(n))$,

Algorithm 4.2: HffO.

Input : An instance $I = (L, \infty, l, w, o)$ of the 2SPOS problem.
Output : The height of the strip used and the solution for I.

4.2.1 Sort the items of I decreasingly by length: $l_1 \geq l_2 \geq \ldots \geq l_n$.
4.2.2 $m \leftarrow 1$; and, let $SOL \leftarrow \{\ \}$ represents the solution for I.
4.2.3 Create the layer m with bottom left corner on point $(0, 0)$.
4.2.4 Put layer m in SOL on point $(0, 0)$.
4.2.5 **for** $i \leftarrow 1$ **to** n **do**
4.2.6 Let k be the layer of minimum height that item i can be packed.
4.2.7 Let (x_k, y_k) be a point in layer of k: x_k represents the x-coordinate where layer k start in the strip, and y_k represents the height this layer.
4.2.8 Pack item i on point (x_k, y_k)
4.2.9 **if** *i is NOT the only item packed on layer k* **then**
4.2.10 Let d_k be the length of layer k that was defined by the first item packed in it.
4.2.11 **else**
4.2.12 $d_k \leftarrow l_i$.
4.2.13 **if** $y_k = 0$ *AND* $x_k + d_k \leq L - l_n$ **then**
4.2.14 $m \leftarrow m + 1$.
4.2.15 Create a new layer m with $(x_m, y_m) \leftarrow (x_k + d_k, 0)$.
4.2.16 Put layer m in SOL on point $(x_m, 0)$.
4.2.17 $y_k \leftarrow y_k + w_i$.
4.2.18 **foreach** *layer* $m \in SOL$ **do**
4.2.19 Let S be the set of items packed in m.
4.2.20 Sort the items of S decreasingly by value of order: $o_1 \geq o_2 \geq \ldots \geq o_{|S|}$.
4.2.21 Compute the center of mass CM_i for each item $i \in S$; and, let d_m be the length of layer m.
4.2.22 Pack the items $(i = 1, \ldots, |S|)$ of S in layer m in such way that the CM_i coincides with $\frac{d_m}{2}$.
4.2.23 Let W be the height of the layer of highest height in SOL.
4.2.24 **return** (W, SOL)

where n corresponds to the number of items in instance I. Note that these algorithms always produce guillotine patterns that are statically stable. In the case of algorithm HffO these patterns also satisfy the order constraint.

The second heuristic presented here consists in a modification of the branch-and-bound algorithm developed by Martello *et al.* (2000) [68] called OneBin. The modified version of this algorithm considers the order and the static stability constraints and is denoted by OneBinOSS (OneBin with Order constraint and Static Stability). The implementation makes use the source code made available by the authors.

The algorithm OneBinOSS works as follows. Let I be an instance of the 2SPOS problem as previously discussed. As this algorithm consider a bin, the initial height W adopted is the sum of the height of all items, that is $W = \sum_{i=1}^{n} w_i$. For each item i denote by a_i the area of item i, and by V the area of the bin.

First, the algorithm sort the items in I decreasingly by the value of order o. Items with the

same value are ordered decreasingly by area. During its execution, the algorithm maintain some sets so as to limit the branch-and-bound ramification. Let P be the set of packed items and N the set of non-packed items. Denote by $C(P)$ the set of corner points and $A(P)$ the area of the envelope [68]. Denote by F the area of the best packing already done.

The sets $C(P)$ and $A(P)$ are updated in each iteration. Each item $j \in N$ is assigned for each corner point $c \in C(P)$ and the algorithm is called recursively, and, if necessary, F has its value updated. The backtracking occurs if $\sum_{i \in P} a_i - (A(P) - V) \leq F$. The order constraint and the static stability are dealt in an easy way. The algorithm always checks if the order and the static stability constraints are violated by an item whenever it is assigned to some point. If true, the backtracking occurs. The heuristic version of the algorithm OneBinOSS simply set the time to a maximum time limit. When this time is reached, the best solution computed is returned, otherwise no solution is provided.

The algorithm OneBinOSS can also be applied to solve the cases without order constraint. In this situation it is only necessary to check if the static stability is violated. This version is denominated by OneBinSS. Both heuristics OneBinOSS and OneBinSS stop when they obtain a packing of the items with all constraints satisfied. So, they are not concerned to generate solutions with minimum height.

The third heuristic is obtained after a simple modification in the algorithm BCut. It consists in solving the integer linear formulation (4.8) considering the discretization at the length direction in the same way it was made at the height direction. As discussed before, this modification on the grid may result in worst solutions. We denominate this heuristic by NonExact.

4.6 Numerical Experiments

The algorithms were implemented in C language and all computational tests were performed in a computer with Intel® Core^{TM} 2 Quad 2.4 GHz processor with 4 GB of RAM memory and *Linux* operating system.

We used the standard framework provided by ILOG® CPLEX® 12 Callable Library (with default parameters) to solve the integer linear formulation in algorithms NonExact and BCut. The CPU time was limited to 3600 seconds for heuristic NonExact and 3600 seconds for algorithm BCut. The algorithms OneBinOSS and OneBinSS had the CPU time limited to 1800 seconds.

4.6.1 The Instances

The computational tests were done in some instances present in the OR-Library [10]. We also generate new random instances.

The instances adapted from the OR-Library are the following: $ngcut01 - ngcut12$ and $cgcut01 - cgcut03$. Such instances were proposed for the Two-dimensional Constrained Knapsack problem considering the oriented case and the case in which the cuts may be non-guillotine and guillotine type, respectively.

The randomly generated instances are divided in three main classes, the first two with 9 instances, and the last one with 6 instances. For all the instances in the first, second and third classes, the strip length is equal to 20, 40 and 60, respectively. Moreover, for the first two classes there are 3 groups of instances each one with 3 instances whose quantity of items is 8, 15 and 20, respectively. In the last class there are only two groups of 3 instances, each one with 8 and 15 items, respectively. The dimensions of each item $i = (l_i, w_i)$ were generated randomly in the closed interval $[0.10L, 0.40L]$. These instances were denominated by rand01, rand02 and rand03 plus the information about the length of the strip and the quantity of items.

A total of 78 instances, with and without order constraints, were considered on the numerical experiments. The order of each item i, for all randomly generated instances, was chosen at random in the set $[1, 2, 3, 4]$. All the mentioned instances are also available at the following url: `http://www.loco.ic.unicamp.br/stability2d/`.

4.6.2 The Algorithms

We combine the algorithm BCut and the heuristics with the objective to reduce the CPU time spent to solve the instances. The most promising combinations are described in Algorithms 4.3 and 4.4.

Algorithm 4.3: First combination of the heuristics with the algorithm BCut.

Input : An instance $I = (L, \infty, l, w, o)$ of the 2SPOS problem.
Output : A solution for I.

4.3.1 **if** *the instance I consider the order constraint* **then**
4.3.2 $\quad (W, S) \leftarrow$ execute HffO(I) and OneBinOSS(I), and returns the best solution found.
4.3.3 **else**
4.3.4 $\quad (W, S) \leftarrow$ execute HffNo(I) and OneBinSS(I), and returns the best solution found.
4.3.5 Create the instance $I' = (L, W, l, w, o)$.
4.3.6 $(W', S') \leftarrow \mathrm{NonExact}(I', S')$.
4.3.7 **if** (W', S') *is empty* **then**
4.3.8 $\quad$ Create the instance $I'' = (L, W, l, w, o)$; $S_{\mathrm{BCut}} \leftarrow \mathrm{BCut}(I'', S)$.
4.3.9 **else**
4.3.10 $\quad$ Create the instance $I'' = (L, W', l, w, o)$; $S_{\mathrm{BCut}} \leftarrow \mathrm{BCut}(I'', S')$.
4.3.11 **return** S_{BCut}.

The Algorithm 4.3 consists in executing the algorithms HffO and OneBinOSS (or HffNo and OneBinSS for the case without order constraints) in order to obtain the first (initial) solution S and the height W of the strip for the instance I. At line 4.3.2 the term "best solution

found" corresponds to a solution of smallest height. The height W is used to create another instance I' in which the strip has length L and height W. Then, the algorithm NonExact use such dimensions to generate the grid of points, and the solution S is used as initial solution (incumbent) for the integer formulation. Finally, the solution and height returned by algorithm NonExact is used as input for the algorithm BCut.

The main difference between Algorithms 4.4 and 4.3 is in the computation of the height. In Algorithm 4.3 the height W was used to create another instance I', and so on. Now, in Algorithm 4.4 the height W computed by algorithms HffO and OneBinOSS (or HffNo and OneBinSS) is used to calculate another height W_1 that is the greatest combination of item heights that is strictly smaller than W.

It remains valid to calculate W_1 in this way, since the horizontal lines of the grid are computed looking for all combinations between the height of the items. Thus, if there is no optimal solution whose height is W, it means that the optimal solution has height smaller than W, that is W_1 or again smaller than W_1, but never a height on the interval $(W_1,\ W]$.

Algorithm 4.4: Second combination of the heuristics with the algorithm BCut.

Input : An instance $I = (L, \infty, l, w, o)$ of the 2SPOS problem.
Output : A solution for I.
4.4.1 **if** *the instance I consider the order constraint* **then**
4.4.2 $(W, S) \leftarrow$ execute HffO(I) and OneBinOSS(I), and returns the best solution found.
4.4.3 **else**
4.4.4 $(W, S) \leftarrow$ execute HffNo(I) and OneBinSS(I), and returns the best solution found.
4.4.5 Let W_1 be the largest combination of item heights with $W_1 < W$.
4.4.6 Create the instance $I' = (L, W_1, l, w, o)$.
4.4.7 $(W', S') \leftarrow \mathrm{NonExact}(I', S')$.
4.4.8 **if** (W', S') *is empty* **then**
4.4.9 Create the instance $I'' = (L, W_1, l, w, o)$; $S_{\mathrm{BCut}} \leftarrow \mathrm{BCut}(I'', S)$.
4.4.10 **else**
4.4.11 Create the instance $I'' = (L, W_1, l, w, o)$; $S_{\mathrm{BCut}} \leftarrow \mathrm{BCut}(I'', S')$.
4.4.12 **return** S_{BCut}.

The main advantage in the use of W_1 instead of W occurs in the pre-processing phase of the solver, but it may also occur in the optimization phase. The reason is if the best packing found by the algorithms HffO and OneBinOSS (or HffNo and OneBinSS) is optimal, the solver returns that the program is infeasible for the grid of points that has height W_1. Consequently, a lot of CPU time can be saved. On the other hand, if the algorithm NonExact computes a solution with height W_1, the same procedure is now repeated for a new height W_2, that is the largest combination of item heights with $W_2 < W_1$, using algorithm BCut.

In Algorithm 4.4 the initial solution is computed by algorithms HffO and OneBinOSS (or HffNo and OneBinSS in the version without order constraints) and cannot be used to start algorithm NonExact or BCut, and so on. This is because the solution obtained by those heuristics

has height W instead of W_1. It is worth mentioning that if the solution of the heuristic NonExact is infeasible, the algorithm BCut can still obtain an optimal solution for the respective instance. And, if the solution returned by algorithm BCut is also infeasible, it is because the initial solution computed by the first heuristics is optimal.

4.6.3 The Results

We first present some information about the instances used on numerical experiments. In Table 4.1, each row has the following data: instance name (Name); length of the strip (L); quantity of items (n); the sum of the height of all items, namely W; the height $W_{\mathcal{A}}$ and the time of algorithm $\mathcal{A}$ spent to obtain a packing, for $\mathcal{A} \in \{\mathrm{HffNo}, \mathrm{HffO}, \mathrm{OneBinSS}, \mathrm{OneBinOSS}\}$.

Observing Table 4.1 the improvement in the height obtained by algorithms HffNo and OneBinSS (HffO and OneBinOSS) over W were in the best situation, respectively, 82.14% for instance rand03^{20}_{20} and 80.95% for instance rand02^{8}_{60} (80.95% for instance rand03^{20}_{20} and 82.25% for instance $ngcut12$). On average, these improvements were 61.81% and 52.36% (55.45% and 51.24%), respectively. These algorithms had CPU time no more than 95 seconds in the worst-case. Only for instances cgcut03, ngcut01, ngcut03, ngcut04, ngcut08 and ngcut09 the height of the solution obtained by algorithm OneBinSS is smaller than that obtained by HffNo. The improvement obtained in this case was 29.01%, on average. For the case with order constraint such improvement was 24.92%, on average, and considering 14 instances. On the other hand, better results were obtained by heuristics HffNo and HffO. The improvement over OneBinSS and OneBinOSS was 31.37% and 35.81%, on average.

It is worth mentioning that the improvement over W obtained by those heuristics allows the solver work with instances that have more items within the time limit imposed.

In order to show the size of the generated models, we presented in Table 4.2 the total number of variables and constraints for the algorithms NonExact and BCut. These numbers were obtained by CPLEX before pre-processing phase. The columns in this table have the following informations: instance name (Name); number of variables (Var) and constraints ($Const$) for algorithms NonExact and BCut, and the difference $Diff_{var}$ (in percentage) on the number of variables between algorithms NonExact and BCut (these informations are for the case without order constraint). The same informations are also presented for the case with order constraint.

Notice that in Table 4.2 some rows have the column $Diff_{var}$ with negative value. It means that algorithm BCut has number of variables (and also constraints) smaller than the ones produced by algorithm NonExact. The reason that it occurs is because Algorithm 4.3 consider the height obtained by the previous algorithm (see lines $4.3.2$, $4.3.4$ and $4.3.6$) as input to the next one. The number of instances in which the column $Diff_{var}$ has negative value corresponds to 14 and 12, respectively, for the case without and with order constraints.

As shown in Table 4.2 the algorithm NonExact returned solutions with a number of variables

Table 4.1: Informations about the instances.

				Without order constraint				With order constraint			
Name	L	n	W	W_{HffNo}	Time (s)	W_{OneBinSS}	Time (s)	W_{HffO}	Time (s)	$W_{\mathrm{OneBinOSS}}$	Time (s)
cgcut01	15	7	23	8	3.00	11	1.00	8	1.00	13	0.00
cgcut02	40	10	151	92	1.00	94	0.00	92	0.00	70	1.00
cgcut03	40	20	551	551	39.00	458	7.00	551	0.00	460	1.00
ngcut01	10	5	24	11	6.00	9	0.00	24	5.00	17	1.00
ngcut02	10	7	29	19	1.00	19	1.00	19	1.00	17	0.00
ngcut03	10	10	36	26	1.00	23	0.00	26	0.00	22	1.00
ngcut04	15	5	11	11	1.00	8	1.00	11	1.00	11	0.00
ngcut05	15	7	25	15	0.00	15	1.00	15	1.00	15	1.00
ngcut06	15	10	46	16	1.00	21	0.00	24	0.00	19	1.00
ngcut07	20	5	18	11	0.00	17	1.00	18	1.00	14	0.00
ngcut08	20	7	29	29	1.00	21	0.00	29	1.00	23	1.00
ngcut09	20	10	63	51	1.00	14	1.00	63	0.00	45	1.00
ngcut10	30	5	55	32	0.00	55	1.00	55	1.00	46	0.00
ngcut11	30	7	61	33	0.00	54	1.00	61	0.00	33	1.00
ngcut12	30	10	169	55	0.00	64	1.00	85	0.00	30	1.00
$\mathrm{rand01}^{8}_{20}$	20	8	37	10	1.00	15	0.00	11	1.00	10	0.00
$\mathrm{rand02}^{8}_{20}$	20	8	38	9	1.00	10	0.00	9	1.00	11	0.00
$\mathrm{rand03}^{8}_{20}$	20	8	35	10	1.00	11	1.00	12	1.00	9	1.00
$\mathrm{rand01}^{15}_{20}$	20	15	44	9	1.00	15	1.00	9	1.00	12	2.00
$\mathrm{rand02}^{15}_{20}$	20	15	36	8	1.00	16	0.00	8	0.00	15	1.00
$\mathrm{rand03}^{15}_{20}$	20	15	37	8	0.00	12	1.00	8	1.00	18	1.00
$\mathrm{rand01}^{20}_{20}$	20	20	59	13	1.00	27	1.00	13	0.00	24	1.00
$\mathrm{rand02}^{20}_{20}$	20	20	60	12	0.00	16	4.00	12	8.00	17	1.00
$\mathrm{rand03}^{20}_{20}$	20	20	56	10	1.00	22	1.00	10	1.00	30	0.00
$\mathrm{rand01}^{8}_{40}$	40	8	51	13	1.00	18	0.00	13	1.00	17	0.00
$\mathrm{rand02}^{8}_{40}$	40	8	46	12	0.00	14	1.00	12	1.00	12	0.00
$\mathrm{rand03}^{8}_{40}$	40	8	55	12	0.00	16	1.00	12	0.00	15	1.00
$\mathrm{rand01}^{15}_{40}$	40	15	91	21	0.00	46	1.00	21	1.00	42	1.00
$\mathrm{rand02}^{15}_{40}$	40	15	88	16	1.00	27	11.00	20	1.00	27	14.00
$\mathrm{rand03}^{15}_{40}$	40	15	90	23	0.00	56	5.00	23	1.00	57	0.00
$\mathrm{rand01}^{20}_{40}$	40	20	124	26	1.00	32	1.00	26	0.00	43	1.00
$\mathrm{rand02}^{20}_{40}$	40	20	122	25	52.00	58	14.00	25	9.00	85	0.00
$\mathrm{rand03}^{20}_{40}$	40	20	133	28	1.00	65	0.00	28	0.00	97	1.00
$\mathrm{rand01}^{8}_{60}$	60	8	75	15	0.00	16	1.00	15	0.00	16	1.00
$\mathrm{rand02}^{8}_{60}$	60	8	84	16	1.00	16	1.00	16	0.00	40	1.00
$\mathrm{rand03}^{8}_{60}$	60	8	68	16	0.00	16	1.00	16	1.00	16	0.00
$\mathrm{rand01}^{15}_{60}$	60	15	140	27	0.00	85	1.00	27	1.00	39	1.00
$\mathrm{rand02}^{15}_{60}$	60	15	123	26	0.00	67	1.00	26	0.00	64	1.00
$\mathrm{rand03}^{15}_{60}$	60	15	129	25	1.00	56	95.00	25	1.00	53	0.00

that are 25.92% and 22.03% smaller than the number of variables used by algorithm BCut, on average, considering the cases without and with order constraint, respectively. Only the non-negative values of $Diff_{var}$ were considered for such calculus. Even with the reduction on the number of variables and constraints, we cannot work with instances with many different item sizes and/or those ones whose dimensions is small compared to the length of the strip.

Tables 4.3 and 4.4 exhibit the results computed using, respectively, the Algorithms 4.3 and 4.4 for the case without order constraints, that is, $o_i = 0$, for $i = 1, \ldots, n$. Tables 4.5 and 4.6 present the results for the case with order constraint, respectively.

In each row of the Tables 4.3 to 4.6 we have the following informations (first presented for the algorithm NonExact): instance name (Name); number of processed nodes (Nodes); number

Table 4.2: Size of the integer formulation for algorithms NonExact and BCut considering Algorithm 4.3.

Name	Without order constraint					With order constraint				
	NonExact		BCut		$Diff_{var}$	NonExact		BCut		$Diff_{var}$
	Var	$Const$	Var	$Const$	(%)	Var	$Const$	Var	$Const$	(%)
cgcut01	1240	4589	1688	7327	26.54	1504	7995	2048	13051	26.56
cgcut02	45318	1475491	62478	3066808	27.47	37408	1706149	51576	3610816	27.47
cgcut03	442442	16200942	707642	87267660	37.48	488844	31370611	781884	171399306	37.48
ngcut01	366	746	606	1438	39.60	869	3636	917	3898	5.23
ngcut02	2413	10327	1833	7375	-31.64	2618	15506	2565	16075	-2.07
ngcut03	4163	17569	3216	14013	-29.45	4576	28608	3696	24417	-23.81
ngcut04	408	794	1208	4883	66.23	726	2079	1568	9305	53.70
ngcut05	1485	4181	3165	16575	53.08	1800	7285	3840	30723	53.12
ngcut06	4496	26329	4816	29800	6.64	6137	57979	6574	65912	6.65
ngcut07	891	2403	2211	7646	59.70	1470	6052	2871	14218	48.80
ngcut08	2820	10168	5058	43302	44.25	3762	20672	6138	81802	38.71
ngcut09	3913	18506	5213	35275	24.94	15224	202829	20284	404628	24.95
ngcut10	671	1418	3311	14213	79.73	1264	4357	6256	42648	79.80
ngcut11	4901	31403	12209	195392	59.86	5945	56678	14819	373372	59.88
ngcut12	26928	630261	28848	709151	6.66	14835	230014	15893	262130	6.66
$rand01^{8}_{20}$	2440	10412	2247	8685	-8.59	2896	18109	2667	15061	-8.59
$rand02^{8}_{20}$	2135	6995	1926	5593	-10.85	2534	11722	2286	9305	-10.85
$rand03^{8}_{20}$	2745	13416	2247	9253	-22.16	2896	18863	2667	16237	-8.59
$rand01^{15}_{20}$	4568	18577	3606	13129	-26.68	5176	31474	4086	22131	-26.68
$rand02^{15}_{20}$	3997	14919	3005	9999	-33.01	4529	25521	3405	16995	-33.01
$rand03^{15}_{20}$	3997	14166	3005	9319	-33.01	4529	23963	3405	15639	-33.01
$rand01^{20}_{20}$	9132	41647	6408	26600	-42.51	10044	70325	10572	75624	4.99
$rand02^{20}_{20}$	8371	37170	6408	26376	-30.63	9207	62415	9691	67113	4.99
$rand03^{20}_{20}$	6849	27826	5607	21322	-22.15	7533	46612	6167	35616	-22.15
$rand01^{8}_{40}$	5193	35626	4487	21405	-15.73	6165	65097	5327	38625	-15.73
$rand02^{8}_{40}$	2645	13775	2564	11876	-3.16	3140	25160	3044	21644	-3.15
$rand03^{8}_{40}$	4039	23130	4487	26843	9.98	4795	41873	5327	48763	9.99
$rand01^{15}_{40}$	19998	246154	21618	272680	7.49	22662	449733	24498	498918	7.49
$rand02^{15}_{40}$	14053	136942	15613	158949	9.99	20825	379501	23137	442107	9.99
$rand03^{15}_{40}$	22220	309382	24020	343039	7.49	25180	570810	27220	633675	7.49
$rand01^{20}_{40}$	34063	474660	36823	526299	7.50	37467	868629	40503	964533	7.50
$rand02^{20}_{40}$	32582	470263	35222	521632	7.50	35838	866196	38742	962046	7.50
$rand03^{20}_{40}$	37025	600156	40025	665874	7.50	40725	1108190	44025	1231040	7.50
$rand01^{8}_{60}$	4245	20684	4805	23953	11.65	5040	37636	5705	43687	11.66
$rand02^{8}_{60}$	5383	35383	6727	46886	19.98	6391	65144	7987	86780	19.98
$rand03^{8}_{60}$	6921	54399	6727	35951	-2.88	8217	100708	7987	65739	-2.88
$rand01^{15}_{60}$	29718	456161	32418	509401	8.33	33678	849918	36738	950083	8.33
$rand02^{15}_{60}$	34671	755838	37821	845293	8.33	39291	1423723	42861	1593493	8.33
$rand03^{15}_{60}$	32420	694502	36020	803746	9.99	36740	1307266	40820	1515344	10.00

of stability cuts (S_{cuts}); number of standard cuts automatically inserted by CPLEX (C_{cuts}); the height of the bin (W), the value of the objective function (OBJ), and the CPU time (in seconds) spent to solve the respective instance. The same informations are also presented for the algorithm BCut.

The entry "–" in tables 4.3 to 4.6 represents that there are no sufficient computer memory available to solve the respective instance. On the other hand, when using the Algorithm 4.4, if the name of the instance appear in boldface means that algorithms NonExact or BCut could not solve it for the initial height considered. In the other cases in which the time limit was

reached (that is, greater or equal than 3600 seconds) the solution returned consists in the best integer solution computed until that moment, if that exists, otherwise the value 0 appears in the columns W and OBJ.

Table 4.3: Performance of Algorithm 4.3 for the case without order constraint.

Name	NonExact						BCut					
	Nodes	S_{cuts}	C_{cuts}	W	OBJ	Time (s)	Nodes	Sb_{cuts}	C_{cuts}	W	OBJ	Time (s)
cgcut01	0	0	5	8	36.010	1.00	3	1	5	8	36.010	2.00
cgcut02	0	0	0	92	3081.239	3612.00	0	0	1319	92	3081.239	3603.00
cgcut03	-	-	-	-	-	-	-	-	-	-	-	-
ngcut01	-	-	-	-	-	-	-	-	-	-	-	-
ngcut02	3473	9	1	13	91.023	224.00	778	0	5	13	91.023	23.00
ngcut03	8677	4	4	16	136.045	3601.00	36694	1	10	16	136.039	3600.00
ngcut04	1	1	16	8	36.013	0.00	1	0	22	8	36.013	1.00
ngcut05	13	0	19	15	120.031	1.00	65	0	13	15	120.031	10.00
ngcut06	4613	0	6	16	136.047	3600.00	5352	1	3	15	120.040	3600.00
ngcut07	0	0	9	11	66.014	0.00	0	0	8	11	66.011	0.00
ngcut08	5212	5	2	19	171.032	258.00	2288	0	3	19	171.032	810.00
ngcut09	-	-	-	-	-	-	-	-	-	-	-	-
ngcut10	0	0	0	32	66.011	0.00	0	0	4	32	66.011	0.00
ngcut11	1520	1	11	33	435.038	398.00	0	0	1363	33	435.038	3601.00
ngcut12	0	1	987	55	1176.091	3603.00	0	0	1022	55	1176.091	3602.00
$\mathrm{rand01}^{8}_{20}$	0	0	3	9	28.003	1.00	0	0	1	9	28.003	0.00
$\mathrm{rand01}^{8}_{20}$	0	1	2	8	21.002	1.00	0	0	5	8	21.002	1.00
$\mathrm{rand01}^{8}_{20}$	0	0	1	8	28.004	0.00	0	0	0	8	28.004	0.00
$\mathrm{rand01}^{15}_{20}$	141	0	8	7	21.013	253.00	17	0	4	7	21.013	29.00
$\mathrm{rand02}^{15}_{20}$	21	1	7	6	15.014	102.00	13	0	14	6	15.014	19.00
$\mathrm{rand03}^{15}_{20}$	26	0	4	6	15.009	50.00	2	0	2	6	15.009	6.00
$\mathrm{rand01}^{20}_{20}$	52	0	4	9	36.031	2867.00	14	0	2	9	36.031	299.00
$\mathrm{rand02}^{20}_{20}$	30	1	1	9	36.030	2396.00	16	0	8	9	36.030	243.00
$\mathrm{rand03}^{20}_{20}$	80	1	7	8	28.029	1188.00	139	0	23	8	28.029	215.00
$\mathrm{rand01}^{8}_{40}$	10	0	3	11	28.003	16.00	4	0	4	11	28.003	7.00
$\mathrm{rand02}^{8}_{40}$	0	0	2	10	10.003	2.00	0	0	1	10	10.003	1.00
$\mathrm{rand03}^{8}_{40}$	18	0	1	12	28.003	16.00	28	0	1	12	28.003	40.00
$\mathrm{rand01}^{15}_{40}$	0	0	1	21	171.055	3600.00	0	0	1	18	120.038	3601.00
$\mathrm{rand02}^{15}_{40}$	11	0	2	16	91.036	3601.00	2	0	1	16	91.036	3601.00
$\mathrm{rand03}^{15}_{40}$	0	0	0	23	210.066	3601.00	0	0	12	23	210.066	3600.00
$\mathrm{rand01}^{20}_{40}$	0	0	0	26	276.136	3600.00	0	1	1	26	276.136	3601.00
$\mathrm{rand02}^{20}_{40}$	0	0	0	25	253.144	3601.00	0	0	0	25	253.144	3600.00
$\mathrm{rand03}^{20}_{40}$	0	0	0	28	325.158	3600.00	0	0	0	28	325.158	3601.00
$\mathrm{rand01}^{8}_{60}$	0	0	0	15	15.001	1.00	0	0	0	15	15.001	1.00
$\mathrm{rand02}^{8}_{60}$	0	0	0	16	28.002	1.00	0	0	0	16	28.002	1.00
$\mathrm{rand03}^{8}_{60}$	0	0	0	13	28.001	1.00	0	0	0	13	28.001	0.00
$\mathrm{rand01}^{15}_{60}$	0	0	0	27	171.047	3600.00	0	1	235	27	171.047	3601.00
$\mathrm{rand02}^{15}_{60}$	0	1	1115	26	231.078	3601.00	0	0	1200	26	231.078	3601.00
$\mathrm{rand03}^{15}_{60}$	0	0	1248	25	210.061	3601.00	0	0	0	25	210.061	3602.00

Table 4.3 shows that the value of the objective function computed by algorithm NonExact is equal to the one computed by algorithm BCut for 82.05% of the instances. For the other instances, the solution returned by algorithm BCut is better than that of NonExact, and the improvement obtained was 29.82% on the best situation (see instance $\mathrm{rand01}^{15}_{40}$). Such improvement obtained by algorithm BCut is related with the way that Algorithm 4.3 consider the height. The number of nodes explored and the time spent by algorithm NonExact was 664 and 1517 seconds, on average. These numbers are equal to 1262 and 1489 seconds considering the

algorithm BCut, on average.

Note that algorithm BCut reached the time limit imposed for 14 against 13 out of 39 instances for the algorithm NonExact. The number of stability cuts applied were of 26 (5) for the algorithm NonExact (BCut) considering all instances. This number corresponds to only 0.74% (0.09%) of the total number of CPLEX cuts applied. More details can be found in Table 4.3.

Table 4.4: Performance of Algorithm 4.4 for the case without order constraint.

Name	NonExact						BCut					
	Nodes	S_{cuts}	C_{cuts}	W	OBJ	Time (s)	Nodes	Sb_{cuts}	C_{cuts}	W	OBJ	Time (s)
cgcut01	-	-	-	-	-	-	-	-	-	-	-	-
cgcut02	0	0	884	0	0.000	3604.00	0	0	1375	0	0.000	3601.00
cgcut03	-	-	-	-	-	-	-	-	-	-	-	-
ngcut01	-	-	-	-	-	-	-	-	-	-	-	-
ngcut02	7849	5	1	13	91.023	257.00	-	-	-	-	-	-
ngcut03	10426	13	2	16	136.038	3600.00	21946	0	4	0	0.000	3601.00
ngcut04	-	-	-	-	-	-	-	-	-	-	-	-
ngcut05	-	-	-	-	-	-	-	-	-	-	-	-
ngcut06	11342	1	1	14	105.035	3600.00	-	-	-	-	-	-
ngcut07	-	-	-	-	-	-	-	-	-	-	-	-
ngcut08	4845	4	12	19	171.033	177.00	-	-	-	-	-	-
ngcut09	-	-	-	-	-	-	-	-	-	-	-	-
ngcut10	-	-	-	-	-	-	-	-	-	-	-	-
ngcut11	-	-	-	-	-	-	-	-	-	-	-	-
ngcut12	0	1	964	0	0.000	3601.00	0	0	999	0	0.000	3601.00
rand01^{8}_{20}	0	0	3	9	28.003	1.00	-	-	-	-	-	-
rand02^{8}_{20}	0	0	2	8	21.003	1.00	-	-	-	-	-	-
rand03^{8}_{20}	7	0	3	8	28.004	2.00	-	-	-	-	-	-
rand01^{15}_{20}	22	1	4	7	21.013	62.00	-	-	-	-	-	-
rand02^{15}_{20}	16	0	1	6	15.014	33.00	-	-	-	-	-	-
rand03^{15}_{20}	31	0	13	6	15.009	25.00	-	-	-	-	-	-
rand01^{20}_{20}	125	0	4	9	36.032	2055.00	-	-	-	-	-	-
rand02^{20}_{20}	100	4	6	9	36.030	1077.00	-	-	-	-	-	-
rand03^{20}_{20}	59	0	11	8	28.029	394.00	-	-	-	-	-	-
rand01^{8}_{40}	22	2	3	11	28.003	19.00	-	-	-	-	-	-
rand02^{8}_{40}	0	0	1	10	10.003	1.00	-	-	-	-	-	-
rand03^{8}_{40}	-	-	-	-	-	-	-	-	-	-	-	-
rand01^{15}_{40}	0	0	1	0	0.000	3600.00	0	0	1	0	0.000	3600.00
rand02^{15}_{40}	29	0	1	14	66.019	2051.00	-	-	-	-	-	-
rand03^{15}_{40}	0	0	1	0	0.000	3600.00	0	0	1	0	0.000	3600.00
rand01^{20}_{40}	0	0	0	0	0.000	3600.00	0	0	0	33	0.000	3601.00
rand02^{20}_{40}	0	0	0	0	0.000	3601.00	0	0	0	0	0.000	3601.00
rand03^{20}_{40}	0	0	0	0	0.000	3600.00	0	0	0	0	0.000	3601.00
rand01^{8}_{60}	-	-	-	-	-	-	-	-	-	-	-	-
rand02^{8}_{60}	-	-	-	-	-	-	-	-	-	-	-	-
rand03^{8}_{60}	-	-	-	-	-	-	-	-	-	-	-	-
rand01^{15}_{60}	0	0	880	0	0.000	3600.00	0	1	235	36	0.000	3600.00
rand02^{15}_{60}	0	0	1116	0	0.000	3601.00	0	0	0	0	0.000	3601.00
rand03^{15}_{60}	0	0	1248	0	0.000	3600.00	0	0	0	0	0.000	3601.00

Observing Table 4.4 the height computed by heuristics HffNo or OneBinSS corresponds to the optimal height for 30.76% of the instances. The height returned by NonExact corresponds to the height of an optimal solution for 38.46% of the instances. Consequently, the use of the Algorithm 4.4 allows 27 out of 39 instances to be solved and avoid the algorithm BCut to be executed.

For the case with order constraint the results are show in Tables 4.5 and 4.6 considering the algorithms 4.3 and 4.4, respectively. As discussed above, the size, and consequently the difficulty increase significantly for the models when compared with the case without order constraint. Notice that the time limit imposed was reached for 43.59% (both algorithms) of the instances in Table 4.5, and for Table 4.6 in 33.34% and 35.89%, considering the algorithms NonExact and BCut, respectively.

Table 4.5: Performance of Algorithm 4.3 for the case with order constraint.

Name	NonExact						BCut					
	Nodes	S_{cuts}	C_{cuts}	W	OBJ	Time (s)	Nodes	Sb_{cuts}	C_{cuts}	W	OBJ	Time (s)
cgcut01	30	0	66	8	36.011	4.00	117	0	135	8	36.011	8.00
cgcut02	0	0	0	70	1596.116	3600.00	0	0	0	70	1596.116	3604.00
cgcut03	-	-	-	-	-	-	-	-	-	-	-	-
ngcut01	0	0	0	11	28.008	0.00	0	0	0	11	28.008	0.00
ngcut02	820	2	319	15	120.035	49.00	761	2	313	15	120.035	54.00
ngcut03	2637	1	1812	16	136.047	3600.00	6993	0	1156	16	136.047	1854.00
ngcut04	0	0	0	8	36.015	0.00	0	0	0	8	36.015	0.00
ngcut05	3	0	60	15	120.041	1.00	41	0	188	15	120.041	13.00
ngcut06	350	2	297	19	190.047	3600.00	100	0	100	16	136.049	3600.00
ngcut07	0	0	19	11	66.011	1.00	0	0	0	11	66.011	1.00
ngcut08	35	0	188	19	171.045	10.00	51	0	957	19	171.043	592.00
ngcut09	0	0	1386	45	990.165	3600.00	0	0	878	45	990.165	3601.00
ngcut10	12	4	49	46	136.019	1.00	27	3	171	46	136.019	8.00
ngcut11	66	2	332	33	435.038	42.00	0	0	573	33	435.038	3600.00
ngcut12	0	0	1766	0	0.000	3606.00	0	0	2190	0	0.000	3600.00
rand01^{8}_{20}	65	2	75	9	28.005	15.00	0	0	18	9	28.005	3.00
rand02^{8}_{20}	0	0	21	8	21.002	2.00	0	0	1	8	21.002	1.00
rand03^{8}_{20}	16	0	14	8	28.005	6.00	0	0	0	8	28.005	1.00
rand01^{15}_{20}	82	2	35	7	21.014	590.00	9	0	14	7	21.014	113.00
rand02^{15}_{20}	125	2	25	6	15.014	574.00	8	0	50	6	15.014	54.00
rand03^{15}_{20}	276	0	72	6	15.010	304.00	1053	0	438	6	15.010	263.00
rand01^{20}_{20}	2	3	12	0	0.000	3600.00	2	0	4	13	78.072	3600.00
rand02^{20}_{20}	4	2	7	0	0.000	3600.00	3	0	10	12	66.062	3601.00
rand03^{20}_{20}	251	3	35	8	28.033	3600.00	1848	0	240	8	28.032	3600.00
rand01^{8}_{40}	12	1	38	11	28.003	71.00	5	0	62	11	28.003	37.00
rand02^{8}_{40}	15	0	44	10	10.003	10.00	0	0	0	10	10.003	4.00
rand03^{8}_{40}	4	2	41	12	28.005	40.00	4	1	27	12	28.003	46.00
rand01^{15}_{40}	0	0	641	0	0.000	3600.00	0	0	636	21	171.056	3601.00
rand02^{15}_{40}	0	0	721	0	0.000	3600.00	0	0	411	20	153.053	3600.00
rand03^{15}_{40}	0	0	0	0	0.000	3600.00	0	0	0	23	210.063	3601.00
rand01^{20}_{40}	0	0	0	0	0.000	3601.00	0	0	0	26	276.133	3600.00
rand02^{20}_{40}	0	0	0	0	0.000	3601.00	0	0	1546	25	253.140	3601.00
rand03^{20}_{40}	0	0	0	0	0.000	3601.00	0	0	0	28	325.154	3602.00
rand01^{8}_{60}	0	0	0	15	15.002	2.00	0	0	0	15	15.002	2.00
rand02^{8}_{60}	0	0	51	16	28.002	21.00	0	0	20	16	28.002	36.00
rand03^{8}_{60}	10	0	693	13	28.001	84.00	0	0	12	13	28.001	28.00
rand01^{15}_{60}	0	0	0	0	0.000	3601.00	0	0	0	27	171.047	3601.00
rand02^{15}_{60}	0	0	0	0	0.000	3601.00	0	0	0	26	231.063	3601.00
rand03^{15}_{60}	0	0	0	0	0.000	3601.00	0	0	0	25	210.050	3602.00

For the case with order constraint the results shown in Tables 4.5 and 4.6 summarize those ones in Tables 4.3 and 4.4, respectively. Observing Table 4.5 the algorithm NonExact could not return any feasible solution (see the value 0.00 in column OBJ) for 12 instances within the imposed time limit, whilst the algorithm BCut returned at least one solution for 11 of these

instances. The reason may be due to the grid of points considered by the last algorithm. On average, the time spent by algorithm NonExact was 1616.82 seconds, whilst algorithm BCut spent 1692.97 seconds. And, the maximum number of stability cuts inserted was 4 and 3, respectively, both for the instance $ngcut10$.

Table 4.6: Performance of Algorithm 4.4 for the case with order constraint.

Name	NonExact						BCut					
	Nodes	S_{cuts}	C_{cuts}	W	OBJ	Time (s)	Nodes	Sb_{cuts}	C_{cuts}	W	OBJ	Time (s)
cgcut01	-	-	-	-	-	-	-	-	-	-	-	-
cgcut02	0	0	0	0	0.000	3602.00	0	0	0	0	0.000	3604.00
cgcut03	-	-	-	-	-	-	-	-	-	-	-	-
ngcut01	0	0	0	11	28.008	0.00	-	-	-	-	-	-
ngcut02	788	3	289	15	120.035	57.00	-	-	-	-	-	-
ngcut03	2534	4	1054	16	136.048	2111.00	-	-	-	-	-	-
ngcut04	0	0	0	8	36.015	0.00	-	-	-	-	-	-
ngcut05	-	-	-	-	-	-	-	-	-	-	-	-
ngcut06	427	1	382	0	0.000	3600.00	313	4	93	0	0.000	3600.00
ngcut07	0	0	21	11	66.011	1.00	-	-	-	-	-	-
ngcut08	43	3	154	19	171.045	12.00	-	-	-	-	-	-
ngcut09	0	0	1286	0	0.000	3600.00	0	0	906	0	0.000	3600.00
ngcut10	-	-	-	-	-	-	-	-	-	-	-	-
ngcut11	-	-	-	-	-	-	-	-	-	-	-	-
ngcut12	-	-	-	-	-	-	-	-	-	-	-	-
rand01^{8}_{20}	146	0	112	9	28.005	18.00	-	-	-	-	-	-
rand02^{8}_{20}	0	0	34	8	21.002	2.00	-	-	-	-	-	-
rand03^{8}_{20}	0	0	26	8	28.004	3.00	-	-	-	-	-	-
rand01^{15}_{20}	19	0	19	7	21.013	302.00	-	-	-	-	-	-
rand02^{15}_{20}	97	0	48	6	15.014	280.00	-	-	-	-	-	-
rand03^{15}_{20}	3108	0	377	6	15.010	871.00	-	-	-	-	-	-
rand01^{20}_{20}	2	2	11	0	0.000	3600.00	2	0	30	0	0.000	3600.00
rand02^{20}_{20}	217	1	7	9	36.032	2738.00	12	0	8	0	0.000	3601.00
rand03^{20}_{20}	2305	0	245	8	28.033	3600.00	-	-	-	-	-	-
rand01^{8}_{40}	10	1	64	11	28.003	74.00	-	-	-	-	-	-
rand02^{8}_{40}	5	0	24	10	10.003	6.00	-	-	-	-	-	-
rand03^{8}_{40}	-	-	-	-	-	-	-	-	-	-	-	-
rand01^{15}_{40}	0	0	506	0	0.000	3601.00	0	0	0	0	0.000	3600.00
rand02^{15}_{40}	0	0	390	5	0.000	3600.00	0	0	369	27	0.000	3601.00
rand03^{15}_{40}	0	0	1222	0	0.000	3601.00	0	0	1142	28	0.000	3601.00
rand01^{20}_{40}	0	0	0	0	0.000	3600.00	0	0	0	0	0.000	3600.00
rand02^{20}_{40}	0	0	0	0	0.000	3601.00	0	0	0	0	0.000	3601.00
rand03^{20}_{40}	0	0	0	0	0.000	3601.00	0	0	0	0	0.000	3602.00
rand01^{8}_{60}	-	-	-	-	-	-	-	-	-	-	-	-
rand02^{8}_{60}	-	-	-	-	-	-	-	-	-	-	-	-
rand03^{8}_{60}	0	0	526	13	28.001	44.00	-	-	-	-	-	-
rand01^{15}_{60}	0	0	0	0	0.000	3601.00	0	0	0	0	0.000	3601.00
rand02^{15}_{60}	0	0	0	0	0.000	3601.00	0	0	0	0	0.000	3602.00
rand03^{15}_{60}	0	0	0	0	0.000	3602.00	0	0	0	0	0.000	3603.00

Now, for Table 4.6 we have that the height computed by algorithm NonExact corresponds to the optimal height for 41.03% of the instances. This number corresponds to 7 out of 39 instances for the heuristics HffO or OneBinOSS.

Comparing the results obtained by Algorithms 4.3 and 4.4, we can state that the last algorithm showed to be much more efficient. So, it can be applied in order to validate (to optimality) the solutions first computed by heuristics and, consequently, to avoid an extensive computation

that is required by an exact approach. With the Algorithm 4.4 only 25 out of 78 instances (see Tables 4.4 and 4.6) were executed by the algorithm BCut.

4.7 Conclusions

In this paper we considered the Two-dimensional Strip Packing problem that have unloading order and static stability constraints. The static stability consider the formulations for the static equilibrium of rigid bodies and the strength of materials. For the Two-dimensional Strip Packing problem with Order constraint we developed an integer linear programming model that was solved by a branch-and-cut algorithm.

We also presented some heuristics. The first one is based on the level-packing algorithm called Hybrid First Fit [24]; the second one was derived from the branch-and-bound approach developed by [68]; and, the last one was a slight modification of the branch-and-cut approach shown in this paper.

The methodology proposed to deal with the static stability is innovative and different from the ones proposed in the literature in the area of cutting and packing problems, since we dealt with the stability in an almost exact way. Our methodology is free-problem in the sense that it can be applied to work consistently well for any packing problem on two-dimensional version. Moreover, with this methodology we can also extend the algorithms to deal with the dynamical stability.

The computational results show that the branch-and-cut approach is consistent and have a good behavior to be applied in practical situations. Although it is limited to solve small to medium-sized instances, where the grid of points is relatively small, the presented heuristics are helpful when integrated with the branch-and-cut approach, mainly in the case of Algorithm 4.4. These heuristics were capable of solving to optimality many instances.

The most promising heuristic in quantity of optimal results computed was that one derived from the branch-and-cut algorithm. However, its CPU time was closer to that spent by the first approach. On the other hand, the heuristic HffNo and HffO was shown to be more eligible for medium- and large-sized instances.

With the presented results we can note the limitations of the CPLEX solver (with default parameters) to solve instances of moderate size for the problem under consideration. Nevertheless, it is worth mention that the solved instances had thousands of variables and constraints showing to be hard to solve.

As future works, we expect to apply these techniques in other problem variants as the three-dimensional case and the Unconstrained Knapsack problem. Then, to extend the static stability for three-dimensional case too, as well as for dynamic stability constraints.

Finally, the algorithms and formulation proposed here can be useful to motivate future research exploring others methods as relaxation methods, probabilistic methods, metaheuristics,

among others, in order to solve large-sized instances and consider other practical constraints as those discussed in [18].

Chapter 5

An Integer Programming Model for the Two-dimensional Strip Packing Problem with Multi-drop and Load Balancing Constraints

Abstract

This paper deals with the Two-Dimensional Strip Packing Problem considering the case in which the center of gravity of the packing must lies on a safety region (load balancing constraint) and the items must satisfy a sequence of loading/unloading (multi-drop requirements). When a subset of items is unloaded, the remaining packing still maintain the center of gravity in the same safety region. We present 0-1 integer linear programming models for this problem and consider only the oriented case. We present two models and a comprehensive performance analysis with several instances. Our computational experiments validate the models and show that they are suitable to deal with problems of small and moderate sizes. Optimal solutions were obtained after few hours of processing time.

5.1 Introduction

Many works on Cutting and Packing problems do not consider practical issues that appear in real-life scenarios, as discussed by Bischoff and Ratcliff (1995) [18]. These issues comprises practical requirements as the load bearing strength of the items [58]; load stability [39, 58, 85];

multi-drop situations [22, 40]; shipment priorities [59, 72]; load balancing (or weight distribution) [54, 59, 72], etc.

We focus on the strip packing problem subject to load balancing constraint with multi-drop requirements. The packing problem can be stated as follows:

TWO-DIMENSIONAL STRIP PACKING PROBLEM (2SP): Given a 2D strip $B = (L, \infty)$ (a bin with length L and infinite height) and a set of n rectangular items, each item i with dimensions (l_i, w_i), determine how to pack all items into the strip B so that the height of the used part of the strip is minimized.

A packing is considered feasible if all the items are packed and arranged orthogonally (i.e. theirs sides are parallel or orthogonal to the strip sides), and there is no overlapping of the items. We consider the oriented case, that is the items can not be rotated. Additionally, other constraints can be included in the problem as we discuss in what follows.

In the typology of Wäscher *et al.* [86] the 2SP problem is called by *Two-dimensional Rectangular Open Dimension Problem*. This problem is $\mathcal{NP}$-hard (see [51]), and with practical applications in industry, as in the cutting of rolls of metal or paper.

The literature have been proposed many heuristic algorithms for 2SP problem. Recent surveys were done by Riff *et al.* (2009) and Ntene and Vuuren (2009) [81, 75]. Hopper and Turton (2001) [51] presented a overview of several metaheuristic strategies. A recent heuristic based on level-packing was presented by Ortmann *et al.* (2010) [77].

In 2009, Kenmochi *et al.* [60] proposed exact branch-and-bound algorithms that improved the strategy developed by [64]. Cintra *et al.* (2008) [27] proposed heuristics based on a column generation approach for the case of guillotine patterns.

In situations where the multi-drop constraints occurs, it is necessary to pack the items within the strip so as to avoid having to unload and re-load a large part of the cargo many times. In this case, we assume that the items are loaded/unloaded by the up-side of the strip. Then, if the cargo follows a route with two or more customers, all items of a particular customer should be available and easy to be removed when the strip arrives at the customer. That is, these items must have free passage to be unloaded. So, we associate to each item i a number d_i that corresponds to its customer, and we require that no item j with $d_j > d_i$ are packed above the given item i.

The multi-drop constraint was explored into the Capacitated Vehicle problem with Loading Constraints [40, 55]. There the variant with multi-drop is called sequential case. Iori *et al.* (2007) [55] presented an exact branch-and-cut algorithm for the two-dimensional version, while Gendrau *et al.* (2008) [40] presented a Tabu search approach for the three-dimensional one. Christensen and Rousøe (2009) [22] present a greedy heuristic that is guided by a tree search for container loading problem subject to multi-drop constraint. They implemented the multi-drop method proposed by Bishoff and Ratcliff [18].

Other common constraints employed by shipping companies are related to load balancing, that is the weight distribution so that the center of gravity (CG) of the cargo (packing) lies close to an "ideal" point or in a region (area) also denominated by *envelope*. In some cases this ideal point (or ideal CG) consists in the geometrical mid-point of the bin/container under consideration. For air-plane and ship (also trucks and trains) cargo, the ideal CG has drastic implications on safety and efficiency like stability and fuel consumption [59, 72]. The center of gravity is defined as the point where all the weight of the object can be considered to be concentrated.

Recent approaches have considered the load balancing constraints obtaining solutions to optimality in an acceptable time. Mongeau and Bès (2003) [72] presented an integer programming formulation for the problem of deciding which items (containers) have to be loaded in plan's compartment satisfying load balance (stability) requirements.

The work of Imai *et al.* (2006) [54] were concerned with the container stowage and load-planning while satisfying the ship stability (it involves the distance between the CG and the ideal CG, known as the metacenter position), list and trim, while minimizing the number of container rehandles. These authors formulate the problem as a multi-objective integer program.

In 2009, Kaluzny and Shaw presented a mixed integer linear programming model for the problem of packing (a subset of) rectangular items in a cargo hold that optimizes the load balance and also consider other constraints as the ability to rotate items and item-specific spacing requirements. Two different objectives were considered. The first one consider to minimize the distance between the center of gravity and the ideal CG, and the second one to maximize the number of the items loaded while the CG must lie within the envelope.

The above papers are concerned with packings where the load balancing is satisfied when all items are packed. In other words, it is not required that the packing satisfy the load balancing when a subset of items is unloaded. A simple scenario where this constraint is important consist in the cases in which a vehicle (ships or even a fleet of trucks) transport goods to customers. In this situation the multi-drop constraints must consider the order in which the customers will be visited. So each customer i (for $i = 1, \ldots, K$) with number d_i means that it will be d_ith customer to be visited in the route. When the first customer is visited his items are unloaded, and, consequently, the packing has its CG moved from its original position.

With this is mind, we consider for the 2SP problem, the load balancing constraint associated with the multi-drop requirement. Thus, not only the final packing, but also each intermediary packing related to a subset of customers non visited in the route must satisfy the load balancing. For the sake of simplicity, this problem is denominated k-2SPMdLb. We present a 0-1 integer linear programming models for this problem.

The paper is organized as follows. In the next section we introduce the integer linear formulation of the 2SP problem. In Section 5.3 we discuss the load balancing and multi-drop constraints used to formulate the k-2SPMdLb problem. The models implemented are shown in

Section 5.4. In Section 5.5 the computational tests are carried out and presented to evidence the real-applicability of the models proposed. Finally, Section 5.6 reports the conclusions and future works.

5.2 The Initial Formulation

The Cartesian plane $\mathbb{R}^2$ is used to represent the packings. Each item is represented by a pair (x, y), where x represents the length and y the height of the object. The position of the item within the packing is represented by its bottom left corner. The pair $(0, 0)$ represents the bottom left corner of the strip.

Let $I = (L, \infty, l_{1,\ldots,n}, w_{1,\ldots,n}, d_{1,\ldots,n}, m_{1,\ldots,n})$ be an instance of the k-2SPMdLb where each item i has dimensions (l_i, w_i), value d_i for Multi-drop constraint and mass m_i, for $i = 1, \ldots, n$. Moreover, we assume that all values in I (except those ones for mass) are integers (if it is not the case, it is just necessary to apply a change of scale).

The problem 2SP is formulated by the binary linear program (5.1). The formulation consider the strip $B = (L, \infty)$ discretized in a grid of points. The distance between each point on the grid on x and y-axis vary according to the grid discretization. Let us assume this distance on x to be represented by c_x. Without loss of generality, the strip at height direction (y-axis) can be discretized observing all combinations between the height of the items.

We denote by $\mathcal{P}$ the set of points obtained from discretization of the strip. Each point $p \in \mathcal{P}$ is represented by a pair $p = (a, b)$. Let $\mathcal{P}_i$ be the set of feasible points where the item i can be packed. And, the set $\mathcal{R}_{ip}$ contains all the points $r \in \mathcal{P}$ that are covered by item i packed on point p, except those points that match the upper and right border of the item.

There are two sets $\mathcal{W}$ and $\mathcal{L}$ used to represent the lines of the grid. The set $\mathcal{W}$ has the lines at the height direction (y-axis), whilst $\mathcal{L}$ has the lines at the length direction (x-axis). Note that an item $i = (l_i, w_i)$ packed on point $p = (a, b) \in \mathcal{P}$ covers all the horizontal lines with height in the set $\mathcal{N}_{ip} = [b,\ b + w_i]$.

The decision variables presented in formulation (5.1) are the following: the variable z_e indicates whether an item cover at least one point $p_e \in \mathcal{P}$ of the horizontal line $e \in \mathcal{W}$. For each item i and point $p \in \mathcal{P}$, the variable $x_{ip} = 1$, if i is packed in p, and $x_{ip} = 0$ otherwise. The variable $y_{ir} = 1$ indicates that the point $r \in \mathcal{P}$ is covered by item i, otherwise $y_{ir} = 0$.

$$
\begin{aligned}
\min \quad & \sum_{e=1}^{|\mathcal{W}|} e z_e & & \\
\textit{subject to}: & & & \\
(i) \quad & \sum_{p \in \mathcal{P}_i} x_{ip} = 1 & & i = 1, \ldots, n. \\
(ii) \quad & \sum_{i=1}^{n} y_{ip} \leq 1 & & \forall\, p \in \mathcal{P}. \\
(iii) \quad & x_{ip} \leq y_{ir} & & i = 1, \ldots, n;\ \forall\, p \in \mathcal{P}_i;\ \forall\, r \in \mathcal{R}_{ip}. \\
(iv) \quad & x_{ip} \leq z_e & & i = 1, \ldots, n;\ \forall\, p \in \mathcal{P}_i;\ \forall\, e \in \mathcal{N}_{ip}. \\
(v) \quad & x_{ip} \in \{0, 1\} & & i = 1, \ldots, n;\ \forall\, p \in \mathcal{P}. \\
(vi) \quad & y_{qi} \in \{0, 1\} & & i = 1, \ldots, n;\ \forall\, q \in \mathcal{P}. \\
(vii) \quad & z_e \in \{0, 1\} & & e = 1, \ldots, |\mathcal{W}|.
\end{aligned}
\tag{5.1}
$$

In formulation (5.1) the objective function aims to minimize the overall height of the strip, and the constraints ensure the feasibility of the packing. In other words, constraints $(5.1.i)$ impose that each item i must be packed exactly once, whilst constraints $(5.1.ii)$ ensure that each point can be covered by only one item; constraints $(5.1.iii)$ impose that all the points in $\mathcal{R}_{ip}$ must be covered by item i whether such item was packed on point p; and, constraints $(5.1.iv)$ ensure that the horizontal lines in the grid covered by any item has to be considered, since they has effect while computing the height of the strip. Constraints $(5.1.v - 5.1.vii)$ ensure the conditions for integrality.

5.2.1 Valid Inequalities

Other constraints can be applied to integer formulation (5.1) in order to speed up the solver and to limit the quantity of combinatorial cases. Then, in an effort to avoid items to "float in mid-air" within the strip the following constraint can be added:

$$
x_{ip} \leq \sum_{\substack{j=1 \\ j \neq i}}^{n} \sum_{q \in \mathcal{S}_{ijp}} x_{jq} \qquad i = 1, \ldots, n;\ \forall\, p \in \mathcal{P}_i. \tag{5.2}
$$

where $\mathcal{S}_{ijp}$ is the set of points in the upper border of item i packed on $p = (a, b)$ in which item $j = (l_j, w_j)$ can be placed to assure that i always have contact with j. Notice that each point $p \in \mathcal{S}_{ijp}$ has abscissa in $(a - l_j,\ a + l_i)$, for $0 \leq a - l_j \leq L$ and $a - l_j \leq a + l_i \leq L$.

5.3 The Multi-drop and Load Balancing Constraints

In this section we describe the multi-drop and load balancing constraints to be added in the formulation (5.1). As mentioned, if there are items of different customers packed in the strip, care has to be taken while loading/unloading. When unloading the items of a given point, they must not be blocked by items that will be unloaded in other points.

The multi-drop requirements are presented in constraints (5.3). The variable u_{pd} indicates whether point $p \in \mathcal{P}$ has value $d \in \mathcal{D}$. The set $\mathcal{D}$ contains all possible values of d_i for $i = 1, \ldots, n$, whilst $\mathcal{D}_j$ have the items whose value is exactly d_j. The value D_{max} corresponds to the highest value within $\mathcal{D}$. Without loss of generality, we also assume that $d_i > 0$ and the set $\mathcal{D}$ is a list sorted decreasingly.

$$
\begin{array}{lll}
(i) & \displaystyle\sum_{d=1}^{D_{max}} u_{pd} \leq 1 & \forall\, p \in \mathcal{P}. \\
(ii) & x_{ip} \leq u_{rd_i} & i = 1,\ \ldots,\ n;\ \forall\, p \in \mathcal{P}_i;\ \forall\, r \in \mathcal{R}_{ip}. \\
(iii) & u_{p'd'} \leq \displaystyle\sum_{d''=d'}^{D_{max}} u_{p''d''} & \forall\, p' \in \mathcal{P};\ \forall\, d',\ d'' \in \mathcal{D};\ such\ that\ d' \leq d'', \\
 & & p'' = \lambda(p'). \\
(iv) & u_{pd} \in \{0, 1\} & \forall\, p \in \mathcal{P};\ \forall\, d \in \mathcal{D}.
\end{array}
\tag{5.3}
$$

where $\lambda(p)$ denotes for each point $p = (a, b) \in \mathcal{P}$ the point $p' = (a, b')$ with $b' = \min\{\beta \in \mathcal{L} \mid \beta > b\}$.

Now, we detail each class of constraints in (5.3). Constraints $(5.3.i)$ ensure that each point of the grid has at most one value of d; constraints $(5.3.ii)$ impose that all the points in $\mathcal{R}_{ip}$ have the same value of d_i; and, constraints $(5.3.iii)$ ensure that any item i with value d_i must be packed over items j with value d_j that satisfy $d_j \leq d_i$. Note that this last set of constraints consider the points of the grid observing only y-axis. If $p' = (a, b')$ then $\lambda(p)$ is a point $p'' = (a, b'')$ where b'' is the next discretization point in $\mathcal{W}$ greater than b'. Constraints $(5.3.iv)$ assure that the variable must be of binary type.

With regard to the load balancing constraint we desire that the center of gravity of the packing (and all sub-packings) lies on an envelope. Giving a packing P, we denote by P_d the packing obtained from P removing items with $d_i < d$. An envelope consists in a rectangular region (that contains the ideal CG) within the strip determined by four coordinates: on the x-axis they are (x_{start}, x_{end}), and on the y-axis, (y_{start}, y_{end}). A packing P is feasible if all packings P_d, for $d \in \mathcal{D}$, have the center of gravity within the envelope. As the objective function aims to minimize the overall height of the strip, we do not have a precise value for the height of the strip, and thus the coordinates of the envelope on y-axis (y_{start}, y_{end}) can be disregard.

$$
\begin{aligned}
(i)\quad & \frac{\sum\limits_{\substack{\forall\, i\ with\\ d_i\in(\mathcal{D}-\cup_{j=1}^{k-1}\mathcal{D}_j)}}\ \sum\limits_{\forall\, p=(a,b)\in\mathcal{P}_i} m_i(a+\frac{l_i}{2})x_{ip}}{\sum\limits_{\substack{\forall\, i\ with\\ d_i\in(\mathcal{D}-\cup_{j=1}^{k-1}\mathcal{D}_j)}} m_i} \geq x_{start} \qquad k=1,\ \ldots,\ D_{max}.\\
(ii)\quad & \frac{\sum\limits_{\substack{\forall\, i\ with\\ d_i\in(\mathcal{D}-\cup_{j=1}^{k-1}\mathcal{D}_j)}}\ \sum\limits_{\forall\, p=(a,b)\in\mathcal{P}_i} m_i(a+\frac{l_i}{2})x_{ip}}{\sum\limits_{\substack{\forall\, i\ with\\ d_i\in(\mathcal{D}-\cup_{j=1}^{k-1}\mathcal{D}_j)}} m_i} \leq x_{end} \qquad k=1,\ \ldots,\ D_{max}.
\end{aligned}
\tag{5.4}
$$

The constraints presented in (5.4) ensure that the center of gravity of the packing and all sub-packings lies on the closed interval $[x_{start}, x_{end}]$. In other words, for each value of $k \in \mathcal{D}$, that is $k = 1, \ldots, D_{max}$, constraints $(5.4.i)$ and $(5.4.ii)$ ensure that the strip with all the items except those ones whose value of d is smaller or equal than k has its center of gravity between the coordinates x_{start} and x_{end}, respectively.

The accurate values of x_{start} and x_{end} depends on the application, and as discussed by Mongeau and Bès (2003) [72] they are set by the master taking into account several uncertainties like the geometric and weight data.

5.4 The Models

The 0-1 integer linear model developed for k-2SPMdLb problem consists in the objective function (5.2) subject to the constraints: (5.1); (5.2); (5.3); and, (5.4).

This formulation depends on the values assumed by c_x, so the exactness of the model is intrinsically associated with the grid of points. We used $c_x = 1$ in the numerical experiments. We called this model by BBound.

We also consider a second model where the strip is also discretized at the length direction (x-axis) considering all combinations between the length of the items. It is called NonBB. Note that this modification may not provide optimal solutions.

The main objective is, then, to compare this two models (BBound and NonBB) considering the solution generated. Also to use the solution computed by model NonBB as initial (feasible) solution for the model BBound.

5.5 Computational Experiments

We coded the algorithms in the C language, and used the standard framework provided by ILOG® CPLEX® 12 Callable Library (with default parameters) to solve the presented models. The main idea of the algorithm is to feed the model NonBB with the sum up of the height of the items as *initial height* for the grid. If a solution was found for this model, it is used as initial solution (incumbent) for model BBound. Otherwise, if no solution was computed for NonBB, we use the same *initial height* for model BBound.

The solver CPLEX uses a branch-and-cut (referred in some contexts as branch-and-bound) algorithm to solve Mixed Integer Linear Programs. As mentioned, we use this solver with its default parameters, so CPLEX automatically manage the optimization process where cuts may be added. For the sake of simplicity, we call these cuts standard cuts. Among these standard cuts, we have clique cuts, cover cuts, disjunctive cuts, flow cuts, Gomory cuts and zero-half cuts. More details can be found in [52].

The experiments were run on a computer with Intel® Core^{TM} 2 Quad 2.4 GHz processor, 4 GB of RAM memory and *Linux* operating system. For each model we set the maximum CPU time to 7200 seconds.

We consider two main classes of instances. The first one consists in instances $cgcut01 - cgcut03$ and $ngcut01 - ngcut12$ present in OR-Library [10]. These instances were used by the Two-dimensional Constrained Knapsack problem considering the guillotine and non-guillotine versions, respectively. Some approaches that used theses instances can be found in [20, 23, 46].

The second class of instances consists in new random instances created in conformity with the k-2SPMdLb problem. It is divided into three groups, each one with 6 instances, and the strip with length of 20, 40 and 60, respectively. In each group there are 3 instances with 8 items and 3 instances with 15 items. The dimensions of each item $i = (l_i, w_i)$ vary between 10% and 40% of the length of the strip. We called these instances by rand01, rand02 and rand03 plus the informations about the length of the strip and the quantity of items.

The ideally center of gravity adopted corresponds to the mid-point of the dimensions of the strip. Thus, we required for numerical experiments that the coordinates of the envelope on the x-axis lies in the closed interval $[0.35L, 0.65L]$, that is: $x_{start} = 35\%L$ and $x_{end} = 65\%L$.

We adopted the value of mass for each item equal to the area of the item. That is, the mass m_i of the item $i = (l_i, w_i)$ is equal to $m_i = l_i w_i$. The integer value d_i, for the Multi-drop constraint, was obtained at random in the interval $[1, \ldots, 4]$. The instances above mentioned are available at the url: `http://www.loco.ic.unicamp.br/balance2d/`.

5.5.1 The Results

We first present some informations about the instances. These informations include the height used to compute the grid of points, the number of variables and constraints for each model.

The number of variables and constraints were obtained before the pre-processing phase of the CPLEX. The following information are shown in each row of the Table 5.1: instance name; length L of the strip; quantity of items (n); height used to compute the grid of points, number of variables ($NVar$) and constraints ($NConst$) for models NonBB and BBound, respectively; and, the difference D_{var} (in percentage) on the number of variables between models NonBB and BBound.

Remember that the height used by the model NonBB corresponds to the sum up of the height of the items, whilst the height for model BBound is the height returned after solving the instance using model NonBB. However, if no solution was found for model NonBB, the height considered for model BBound is the same used by NonBB. As a result some instances present in Table 5.1 have the column D_{var} with negative value.

Observing Table 5.1, the number of instances where the model BBound outperforms (has less variables and constraints than) NonBB is equal to 11 out of 33. It represents 33.34% of the instances considered on numerical experiments. And, the difference between the number of variables is of 60.16%, on average. On the other hand, the model NonBB has less variables and constraints than BBound only for cases in which the instances on model NonBB can not be solved or no feasible solution was found within the time limit imposed. Anyway note that the instances are hard to be solved, since they have thousands of variables and constraints for both models.

Table 5.2 illustrates the results obtained by the solver considering the previous models. The following informations are shown in each row of this table: instance name; the number of branching nodes visited (Nodes), the number of standard cuts generated by CPLEX (Cuts), the height W and the center of gravity (CG) of the strip, the optimal value (OPT) computed, and, the time spent (in seconds) to solve the respective instance considering the model NonBB. These informations are also presented for model BBound.

Some instances in Table 5.2 have the entry "–". This means that there are no sufficient computer memory available. However, there are cases where the time limit for each model was reached, but at least one feasible solution was found, in this case, we present the informations about this solution. On the other hand, when the value 0 appears at same time in columns W, CG and OPT means that the optimization processes was interrupted because the time limit imposed was reached and no feasible solution was found.

Observing Table 5.2 we note that the number of branching nodes visited by NonBB and BBound was of 80 and 82, on average. As a result, the time spent, on average, was of 4559.79 and 4168.82 seconds, respectively. Despite the difference in number of nodes visited and time spent, both models returned solutions with equal values (see column OPT) for 72.73% of the instances.

In model BBound better solutions were computed compared to the model NonBB. However, as presented in Table 5.1, the height used by BBound to compute the grid of points was different

Table 5.1: Informations about the 0-1 integer models NonBB and BBound.

Name	L	n	NonBB			BBound			D_{var}
			$Height$	$NVar$	$NConst$	$Height$	$NVar$	$NConst$	(%)
cgcut01	15	7	23	4324	31641	8	2048	13057	-52.64
cgcut02	40	10	151	85731	5336621	151	118203	11116031	27.47
cgcut03	40	20	551	571419	37730292	551	913959	206733048	37.48
ngcut01	10	5	24	1185	5695	17	1441	7480	17.77
ngcut02	10	7	29	4466	30404	14	2394	14563	-46.39
ngcut03	10	10	36	7812	51212	16	3856	24725	-50.64
ngcut04	15	5	11	726	2085	8	1568	9311	53.70
ngcut05	15	7	25	3000	14281	15	3840	30729	21.88
ngcut06	15	10	46	15502	166383	26	9386	98142	-39.45
ngcut07	20	5	18	1890	8394	11	2871	14224	34.17
ngcut08	20	7	29	4617	26912	19	6138	81808	24.78
ngcut09	20	10	63	22021	303161	45	29341	604217	24.95
ngcut10	30	5	55	1501	5650	41	5474	35616	72.58
ngcut11	30	7	61	10865	141755	38	17374	476598	37.46
ngcut12	30	10	169	104315	6421316	169	111755	7210196	6.66
rand01^{8}_{20}	20	8	32	10860	125531	8	2667	16904	-75.44
rand02^{8}_{20}	20	8	37	11946	155943	19	6477	70161	-45.78
rand03^{8}_{20}	20	8	35	11946	140044	8	2667	16243	-77.67
rand01^{15}_{20}	20	15	45	27821	246956	45	29283	265831	4.99
rand02^{15}_{20}	20	15	49	30409	258660	49	32007	278263	4.99
rand03^{15}_{20}	20	15	46	28468	249869	16	10215	80384	-64.12
rand01^{8}_{40}	40	8	57	31703	915477	15	8371	114289	-73.60
rand02^{8}_{40}	40	8	48	25880	715005	14	7610	104308	-70.600
rand03^{8}_{40}	40	8	51	28638	879259	17	9893	186008	-65.45
rand01^{15}_{40}	40	15	113	127437	5005222	113	145627	6140923	12.49
rand02^{15}_{40}	40	15	85	99461	2989601	85	107519	3331913	7.49
rand03^{15}_{40}	40	15	92	108274	3477828	92	117046	3872751	7.49
rand01^{8}_{60}	60	8	85	58078	3239488	85	81011	5445643	28.31
rand02^{8}_{60}	60	8	86	37830	1245490	86	44499	1590066	14.99
rand03^{8}_{60}	60	8	75	58716	3343548	75	71883	4567162	18.32
rand01^{15}_{60}	60	15	142	224114	15284959	142	249002	17640197	10.00
rand02^{15}_{60}	60	15	148	254456	16462497	148	277576	18425587	8.33
rand03^{15}_{60}	60	15	150	253506	16290909	150	281658	18894081	10.00

of that used by model NonBB for 57.58% of the instances. Moreover, the model BBound found better results (see column OPT) than the latter one for 12.5% of the instances as shown in Table 5.2. The improvement obtained was of 39.17%, on average, and 81.70% on the best situation (see instance rand02^{8}_{20}).

With the time limit imposed, only 13 optimal solutions were found using model BBound. And, the number of optimal instances for NonBB were 11 out of 33 instances. Besides that, the number of standard cuts applied, on average, was of 308 and 189 for models NonBB and BBound, respectively. Other details can be found in Tables 5.1 and 5.2.

Table 5.2: Solutions computed for models NonBB and BBound.

Name	NonBB						BBound					
	Nodes	$Cuts$	W	CG	OPT	Time (s)	Nodes	$Cuts$	W	CG	OPT	Time (s)
cgcut01	118	344	8	(6.91, 3.86)	36.011	116.00	174	222	8	(6.91, 3.86)	36.011	8.00
cgcut02	-	-	-	-	-	-	-	-	-	-	-	-
cgcut03	-	-	-	-	-	-	-	-	-	-	-	-
ngcut01	20	68	17	(4.77, 8.99)	66.021	1.00	0	10	17	(4.77, 8.99)	66.021	0.00
ngcut02	274	574	14	(4.82, 6.14)	105.033	51.00	52	134	14	(4.82, 6.14)	105.033	5.00
ngcut03	1024	289	16	(4.55, 7.85)	136.051	2083.00	482	82	16	(4.55, 7.85)	136.051	85.00
ngcut04	0	0	8	(6.92, 4.25)	36.015	0.00	0	0	8	(6.92, 4.25)	36.015	0.00
ngcut05	493	333	15	(6.45, 7.43)	120.041	14.00	0	157	15	(6.45, 7.43)	120.041	18.00
ngcut06	11	145	26	(8.40, 11.19)	351.080	7200.00	174	167	24	(6.44, 11.56)	300.080	7200.00
ngcut07	13	64	11	(9.92, 4.06)	66.011	1.00	0	1	11	(9.92, 4.06)	66.011	1.00
ngcut08	280	626	19	(8.65, 9.38)	171.043	28.00	540	118	19	(8.65, 9.38)	171.043	2873.00
ngcut09	-	-	-	-	-	-	-	-	-	-	-	-
ngcut10	0	0	41	(13.20, 19.86)	105.023	0.00	0	8	41	(13.20, 19.86)	105.023	3.00
ngcut11	4	844	38	(16.70, 15.42)	595.047	7200.00	0	626	38	(16.70, 15.42)	595.047	7208.00
ngcut12	-	-	-	-	-	-	-	-	-	-	-	-
rand01$^{8}_{20}$	56	41	8	(8.97, 3.45)	28.007	154.00	0	23	8	(8.97, 3.45)	28.007	2.00
rand02$^{8}_{20}$	15	1863	19	(9.40, 5.88)	153.017	7201.00	511	216	9	(9.65, 3.90)	28.005	811.00
rand03$^{8}_{20}$	0	18	8	(9.10, 3.52)	28.004	12.00	0	17	8	(10.32, 3.52)	28.004	3.00
rand01$^{15}_{20}$	0	162	0	(0.00, 0.00)	0.000	7200.00	0	101	0	((0.00, 0.00)	0.000	7200.00
rand02$^{15}_{20}$	0	33	0	(0.00, 0.00)	0.000	7201.00	0	35	0	(0.00, 0.00)	0.000	7202.00
rand03$^{15}_{20}$	0	569	16	(9.72, 5.48)	120.047	7200.00	20	64	16	(9.72, 5.48)	120.047	7200.00
rand01$^{8}_{40}$	0	829	15	(19.46, 5.12)	66.004	7203.00	0	1094	15	(19.46, 5.12)	66.004	1276.00
rand02$^{8}_{40}$	0	1031	14	(19.48, 5.51)	55.005	7204.00	418	741	11	(18.49, 5.01)	28.005	7200.00
rand03$^{8}_{40}$	0	1078	17	(21.60, 6.89)	91.016	7208.00	0	1641	13	(18.88, 5.56)	45.009	7200.00
rand01$^{15}_{40}$	0	0	0	(0.00, 0.00)	0.000	7249.00	0	0	0	(0.00, 0.00)	0.000	7277.00
rand02$^{15}_{40}$	0	0	0	(0.00, 0.00)	0.000	7214.00	0	0	0	(0.00, 0.00)	0.000	7215.00
rand03$^{15}_{40}$	0	0	0	(0.00, 0.00)	0.000	7224.00	0	0	0	(0.00, 0.00)	0.000	7215.00
rand01$^{8}_{60}$	0	0	0	(0.00, 0.00)	0.000	7214.00	0	0	0	(0.00, 0.00)	0.000	7345.00
rand02$^{8}_{60}$	0	0	0	(0.00, 0.00)	0.000	7204.00	0	0	0	(0.00, 0.00)	0.000	7210.00
rand03$^{8}_{60}$	0	0	0	(0.00, 0.00)	0.000	7207.00	0	0	0	(0.00, 0.00)	0.000	7247.00
rand01$^{15}_{60}$	0	0	0	(0.00, 0.00)	0.000	7227.00	0	0	0	(0.00, 0.00)	0.000	7331.00
rand02$^{15}_{60}$	0	0	0	(0.00, 0.00)	0.000	7214.00	0	0	0	(0.00, 0.00)	0.000	7345.00
rand03$^{15}_{60}$	0	0	0	(0.00, 0.00)	0.000	7204.00	0	0	96	(0.00, 0.00)	0.000	7210.00

From these experiments, we can state that the model BBound requires much more CPU time compared to model NonBB. However, the combination of these two models seems to be the best choice, since the solutions computed by NonBB can be used as input for BBound, and avoid an expensive computation required by the last one.

5.6 Conclusions and Future work

This paper deals with the Two-dimensional Strip Packing problem subject to the load balancing and the multi-drop constraints. This means that the entire packing as well as the intermediary packings must satisfy the load balancing constraints.

After several numerical experiments, we note that the accuracy of the results depends of the grid of points used in the formulation. Consequently, if the grid of points is dense, the CPU time increases accordingly.

Several others instances were solved to optimally with the two presented models within the time limited imposed of 7200 seconds. These show that the models are suitable and consistent to represent practical situations, although they are limited to solve instances where the number of points in the grid is relatively small.

In conclusion, the models presented in this paper allows for a solution of the Strip packing problem under a realistic scenarios (with practical constraints), however considerable room for improvement and extension of the models remains.

Chapter 6

Heuristics for Two-Dimensional Irregular Cutting and Packing Problems

Abstract

There are several papers that deal with the two-dimensional version of cutting and packing problems for regular/rectangular shapes. However packing problems with irregular item shapes have not received as much attention as problems with regular shapes. In this work we propose algorithms for cutting and packing problems for objects with irregular shapes based on the computation of No-Fit polygons. We present a GRASP based heuristic for the 0/1 version of the Knapsack Problem, and a heuristic for the unconstrained version of the Knapsack Problem. This last heuristic is divided on two steps: first pack irregular items in rectangles and then use the rectangles as regular items to be packed in the bin. This algorithm is then combined with a column generation algorithm to solve the Cutting Stock problem with irregular objects. The algorithms proposed found optimal solutions for several of the tested instances within a reasonable runtime. For other instances, the algorithms obtained solutions for almost all instances with occupancy rates above 90% with relatively fast execution time.

6.1 Introduction

In this paper we investigate cutting and packing problems with items of irregular shapes. These problems have several practical applications including: garment manufacturing, sheet metal cutting, furniture making; shoes manufacturing ([16, 44]). Naturally, packing items inside a bin can be seen as the same as cutting the bin in order to obtain the items. Thus, from now on, we

use the term packing or cutting with the same meaning. The problems of interest in this paper are the following:

TWO-DIMENSIONAL IRREGULAR 0/1 KNAPSACK PROBLEM (2KPI-(0/1)): We are given a bin B and a list T of n items, each item i may have an irregular or regular shape, and has value v_i, for $i = 1, \ldots, n$. The objective is to determine how to cut some items of T in order to maximize the total value of the items that are produced from the bin. In this problem, at most one item of each shape can be produced, so some items may not occur.

TWO-DIMENSIONAL IRREGULAR UNCONSTRAINED KNAPSACK PROBLEM (2KPI): In this problem we have the same input as in the above problem. But now one item of T can be cut several times from the bin. The objective is the same, but many copies of one item can be produced (again some items may not occur), hence the term *unconstrained*.

TWO-DIMENSIONAL IRREGULAR CUTTING STOCK PROBLEM (2CSI): We are given an unlimited quantity of bins B and a list T of n items, with irregular or regular shapes, each item i with demand d_i for $i = 1, \ldots, n$. The objective is to determine how to cut the smallest number of bins B so as to produce d_i units of each item i.

For all the above mentioned problems, the items are represented by polygons (that may contain holes), and may have irregular shape. The bins are rectangles with length L and width W. The items are represented by convex and non-convex polygons, with oriented edges, and any curve is approximated by a sequence of (exterior) tangent segments. A feasible placement of the items in the bin consists in a placement where no two items overlap and all of them fit inside the bin. All the problems above mentioned are $\mathcal{NP}$-hard [37].

According to Bennell and Oliveira (2006) [15] these problems are referred to as *nesting problems*, that are problems where more than one item of irregular shape must be placed in a configuration with the other item(s) in order to optimize an objective function.

Albano and Sapuppo (1980) [2] presented a nesting algorithm combined with the no-fit polygons that aims to reduce the geometric complexity of the nesting process. The no-fit polygon has become an increasingly popular and efficient option for dealing with the geometry of packing problems with irregular shapes. The purpose of no-fit polygons is to present regions of feasible placement of items such that they do no overlap and are placed inside the bin. The term no-fit polygon was introduced by Adamowicz and Albano (1976) [1]. Oliveira *et al.* (2000) [76] also use no-fit polygons solution to solve nesting problems.

Daniels and Milenkovic (1997) [30] proposed exact and approximated algorithms for nesting problems restricted to the case where we have at most seven different items, and items and the bin have convex shape. Approaches based on linear programming can be found in [14];

Hopper (2000) [69] presented a genetic algorithm in combination with the bottom-left and the bottom-left-fill approaches for both orthogonal and irregular nesting problems. Burke *et al.* (2007) [19] presented an overview of several techniques for the computation of no-fit polygon,

and then, proposes a complete and robust no-fit polygon algorithm. Their approach is based on a robust orbital method that involves two geometric stages. A recent survey covering the core geometric methodologies employed in cutting and packing of irregular shapes was done by Bennell and Oliveira (2008) [15].

In 2010, Martins and Tsuzuki [70] proposed a heuristic based on simulated annealing for the problem of minimizing the waste while packing a set of irregular items inside a bin, where rotation of items are allowed. Other approaches especially metaheuristics can be found in [34, 63];

For the problem of packing regular shapes (rectangles) inside a rectangle, Cintra *et al.* (2008) [27] present dynamic programming algorithms for the 2KP problem with guillotine cuts and its variants that allows rotation of the items and cuts in at most k stages. Puchinger and Raidl (2007) [78] presented a branch-and-price algorithm for the 2CS problem where the cuts have to be done in at most k stages. Cintra *et al.* (2008) [27] also consider the 2CS problem restricted to cuts of guillotine type. Their approach is a column generation method that uses the dynamic programming algorithm proposed for the 2KP problem in order to generate the columns. Other approaches are discussed in [11, 17, 20, 41, 46, 48].

In this paper we propose the following:

- For the 2KPI-(0/1) problem: a GRASP algorithm that uses the robust no-fit polygon strategy proposed by Burke *et al.* [19] to compute the no-fit polygons (NFP) between each pair of items and on each possible rotation, and, then, to ensure feasible layouts; and, a search algorithm developed by Adamowicz and Albano [1] to compute the best arrangement of an item inside a packing with the objective to minimize the area of the rectangular and convex closures and also respect the dimensions of the bin.

- For the 2KPI problem: a strategy that combines the approaches developed by Burke *et al.* [19] and Adamowicz and Albano [1] to generate rectangles with high occupation. Then, these rectangles are used as items for the exact dynamic programming approach developed by [27] to deal with the regular (rectangular items) and the guillotine version of the 2KP problem. We combine such algorithm for the 2KP problem with the *reduced raster points* of Scheithauer (1997) [83].

- For the 2CSI problem: we use the column generation based approach presented in [27], using the algorithm proposed for the 2KPI problem to generate new columns.

This paper is organized as follows. In Section 6.2, we discuss about the 2KPI-(0/1) problem. We outline the way the no-fit polygon is computed and the GRASP based algorithm developed to solve this problem. Sections 6.3 and 6.4 present the algorithm to solve the 2KPI problem and outlines the column generation approach for the 2CSI problem. In Section 6.5 the numerical experiments are reported in order to show the efficiency of the proposed algorithms. Finally, in Section 6.6 some concluding remarks are presented.

6.2 2KPI-(0/1) Problem

First, we discuss about the geometrical questions that arise in problems with irregular shapes. Naturally, the most visible attribute of nesting problems is the geometry.

6.2.1 The No-Fit Polygon Generation

Some approaches handled in the literature include the raster method, direct trigonometry, the no-fit polygon and the phi-function. The most promising strategy consists in the computation of the no-fit polygon (NFP) [15].

The NFP can be stated as the polygon obtained by combining two items in such a way that: (i) the interior of the NFP represents the relative positions of these two items that result in overlap; (ii) the boundary consists in the touching positions between such items; and, (iii) the exterior represents their separation. In this paper we codify the robust approach developed by Burke *et al.* (2007) [19]. We call this algorithm by NoFitP.

This approach is used to calculate the no-fit polygon NFP$_{AB}$ (orbiting item B on its reference point around item A) for two items A and B. It is divided on two phases. In the first phase the item B slides around the item A to create the outer path of the no-fit polygon of these two shapes. The second one consists in find the remaining paths (internal holes) of the no-fit polygon that were not found in the first phase. Details about the implementation of the algorithm NoFitP can be found in [19].

The algorithm NoFitP allows to compute all feasible positions between two items, when they do not overlap and they are touching each other. However, a simple packing can involve several items. First, assume that each item i has a set of allowable rotations $\Theta = \{\theta_1, \ldots, \theta_m\}$. Then, it is just necessary to use the algorithm NoFitP to compute the no-fit polygons between all the pairs of items and for each allowable rotation.

Therefore, when a new item is packed, we already know the no-fit polygon, denoted by $NFP(i, j, \theta_i, \theta_j)$, of the new item i, whose rotation is θ_i, in relation to each item already packed j, whose rotation is θ_j. In other words, for a set of items already packed $\{j_1, \ldots, j_k\}$ and a new item i to be packed, becomes necessary to generate the NFP obtained by $\bigcup_{q=1}^{k} NFP(i, j_q, \theta_i, \theta_{j_q})$.

6.2.2 The Search Algorithm

The search algorithm proposed by Adamowicz and Albano (1976) [1] is adapted here to compute the best position inside the bin to pack an item. In our version the objective is to search a point that minimizes the area of the rectangular and convex closures. The dimensions of the bin must also be considered. We denominate such version by ExSEARCH.

We denote by *rectangular area* of a set of packed items as the area of the rectangular closure where all these items are packed. The *convex area* corresponds to the area of the convex closure.

Next, we describe the idea of the algorithm ExSEARCH considering two items A and B as input. Assume that the no-fit polygon NFP$_{AB}$ was previously computed by algorithm NoFitP. The main idea is to execute a search on all vertices of NFP$_{AB}$. For each vertex in NFP$_{AB}$, the item B is, then, translated considering its reference point and the set of allowable rotations, so the area of rectangular and convex closures are computed. Thus, the current best solution is updated whenever a better solution is computed. It is important to mention that the dimensions of the bin are also considered for such computation. When holes are found in NFP$_{AB}$ the algorithm first executes the search in such holes.

To deal with a set of items, the algorithm ExSEARCH considers two by two items. The first two items are, then, combined in order to create a single item. Then, the algorithm consider this new single item and other item of the set. This process is repeated always considering two items: the item created by the union of the items already packed and another item of the set.

6.2.3 Packing Single Items

Now we describe the algorithm that uses the algorithms NoFitP and ExSEARCH with the objective to pack an item in a bin that may already contain items. This algorithm is referred to as PackS and is presented in Algorithm 6.1.

The algorithm PackS has as input an item i to be packed, the set of allowable rotations Θ that this item may assume, where $|\Theta| = m$, the dimensions of the bin (L, W), and a (partial) packing P of items. It returns -1 if the item i can not be packed inside the bin of dimensions (L, W). Otherwise, it returns the point (x, y) where item i can be packed and the rotation θ_i considered for such packing.

Observing algorithm PackS, the variable $flag$ indicates whether item i can be packed inside the bin. The main loop at lines $6.1.3 - 6.1.15$ consider all m rotations of item i in order to compute a feasible packing that generates the rectangular closure of smallest area. Notice that at line $6.1.7$ the algorithm calculates the no-fit polygon between item i, with rotation θ_i, and each item j packed in P on rotation θ_j. The no-fit polygon in variable nfp consists of the union of the individual no-fit polygons i and each j.

The algorithm ExSEARCH is used at line $6.1.8$ to find the best point to pack item i at rotation θ_i. As mentioned, the algorithm finds a point that minimizes the area of the rectangular closure, stored in the variable $areaR$, and the convex closure, stored in the variable $areaC$, and that respect the dimensions (L, W) of the bin. If ExSEARCH returns $flag'$ as true, then a new solution was found so that item i is packed on point (x', y') at rotation θ_i, and the variables may have their values updated (see lines $6.1.9 - 6.1.15$).

Algorithm 6.1: PackS.

Input : item i; set of rotations Θ for i; dimensions of the bin (L, W); packing P.
Output : point (x, y) and angle θ to pack item i inside bin.

6.1.1 $flag \leftarrow false$.
6.1.2 $(areaR, areaC) \leftarrow (0, 0)$.
6.1.3 **foreach** $\theta_i \in \Theta$ **do**
6.1.4 $\quad flag' \leftarrow false$.
6.1.5 $\quad nfp \leftarrow \{\ \}$.
6.1.6 $\quad$ **foreach** *item* $j \in P$ **do**
6.1.7 $\quad\quad nfp \leftarrow nfp \cup \mathrm{NoFitP}(i, j, \theta_i, \theta_j)$.
6.1.8 $\quad (x', y', flag') \leftarrow \mathrm{ExSEARCH}(nfp, i, \theta_i, L, W)$.
6.1.9 $\quad$ **if** $flag'$ **then**
6.1.10 $\quad\quad$ Let $areaR'$ be the area of the rectangular closure for $P \cup \{i, (x', y'), \theta_i\}$.
6.1.11 $\quad\quad$ Let $areaC'$ be the convex closure for $P \cup \{i, (x', y'), \theta_i\}$.
6.1.12 $\quad\quad$ **if** $areaR = 0$ **OR** $areaR' < areaR$ **OR** $(areaR' = areaR$ **AND** $areaC' < areaC)$ **then**
6.1.13 $\quad\quad\quad (x, y, \theta) \leftarrow (x', y', \theta_i)$.
6.1.14 $\quad\quad\quad areaR \leftarrow areaR'$; $areaC \leftarrow areaC'$.
6.1.15 $\quad\quad\quad flag \leftarrow true$.
6.1.16 **if** $flag$ **then**
6.1.17 $\quad$ **return** $\{1, (x, y), \theta\}$.
6.1.18 **return** $\{-1, (-1, -1), -1\}$.

6.2.4 GRASP Algorithm for the 2KPI-(0/1) Problem

Now, we present the GRASP based heuristic to solve the 2KPI-(0/1) problem. We consider that the value of each item corresponds to its area and as mentioned each item is represented by a polygon. The first step to construct our heuristic is to generate an initial solution (by some greedy algorithm). Then, a local search algorithm that aims to improve the initial solution is repeatedly applied.

The greedy randomized algorithm used to generate the initial solution constructs a solution where the items are packed one by one. This algorithm is described in Algorithm 6.2 and it is called by IniSOL.

In each iteration of lines $6.2.2 - 6.2.12$, the algorithm IniSOL chooses a random subset of items (line $6.2.3$) to generate the set I'. From the items in I', the algorithm selects an item with the best contribution for the partial solution S. That is, all feasible items are tested (packed) considering the set of rotations Θ. Notice that algorithm PackS is used at line $6.2.5$ to pack each item i. Next, if some item i can be packed, the cost to add i into S is computed (line $6.2.7$). This cost is related to the total area of items packed in the bin and with the area of the rectangular

Algorithm 6.2: IniSOL.

Input : set of items I; set of rotations Θ for items in I; dimensions of the bin (L, W); percentage p of items to be chosen at random.

Output : solution (packing) S; set I_1 with the items that are in S; set I_2 with the items that are not in S.

6.2.1 $S \leftarrow \{\ \}$.
6.2.2 **while** $I \neq \{\ \}$ **do**
6.2.3 $I' \leftarrow$ choose $p\%$ of the items from I at random.
6.2.4 **foreach** $i \in I'$ **do**
6.2.5 $(\mathit{flag}, x, y, \theta) \leftarrow \mathrm{PackS}(i, \Theta, L, W, S)$.
6.2.6 **if** $\mathit{flag} \neq -1$ **then**
6.2.7 Compute the cost to add i into S.
6.2.8 **else**
6.2.9 $I' \leftarrow I' - \{i\}$; $I \leftarrow I - \{i\}$; $I_2 \leftarrow I_2 \cup \{i\}$.
6.2.10 **if** $I' \neq \{\ \}$ **then**
6.2.11 Select $i \in I'$ of smallest cost and get the rotation θ and the point (x, y) of such item.
6.2.12 $I \leftarrow I - \{i\}$; $I_1 \leftarrow I_1 \cup \{i\}$; $S \leftarrow S \cup \{i, \theta_i, (x, y)\}$.
6.2.13 **return** (S, I_1, I_2).

closure. Otherwise, item i is removed from set I and added into I_2 (line $6.2.9$).

Observing lines $6.2.10 - 6.2.12$ note that the set I' have all the items that can be added into S. In the case where two or more items are in I', the item of smallest cost, that is with the best occupation of the bin. Ties are broken selecting an item which leads to the rectangular closure of smallest area. The algorithm ends when the set I is empty.

The solution S (that is a packing of items) returned by algorithm IniSOL is the initial solution for the GRASP based algorithm. The next step is, thus, to improve such solution. The Algorithm 6.3, denoted by LocSEARCH, is a local search procedure that aims to improve an initial solution. We assume that a solution consists in a sequence of items $(i_1, \ldots, i_s)$ packed by the algorithm PackS. The *neighbors* of a solution S, considered by the algorithm LocSEARCH, are obtained by applying simple modifications into S as insertions or permutations of items. Note that the neighbors of a solution are also solutions. The local search procedure finishes when a solution can not be improved.

On each iteration of the lines $6.3.1 - 6.3.6$, the algorithm generates x neighbors of solution S using the sets I_1 and I_2. The neighbors are obtained by insertions and permutations of items in S and these lists. We consider three different types of neighbors, each one selected with a given probability. The three types of neighbors are the followings:

1. A neighbor is obtained by permuting two items of I_1. This new order of the items may produce a more compact packing leaving space for some item $j \in I_2$ to be packed in I_1.

Algorithm 6.3: LocSEARCH.

Input : solution S; set I_1 with the items that are in S; set I_2 with the items that are not in S; dimensions of the bin (L, W); number of iterations x; probabilities (q_1, q_2, q_3) for each type of neighbor.

Output : solution S possibly improved.

6.3.1 **while** S *can be improved* **do**

6.3.2 **for** $i \leftarrow 1$ **to** x **do**

6.3.3 Choose a type t of neighbor at random considering probabilities (q_1, q_2, q_3).

6.3.4 Compute a neighbor of the solution S considering type t, and sets I_1 and I_2.

6.3.5 **if** *there is at least one neighbor* S' *of* S *that was improved* **then**

6.3.6 $S \leftarrow S'$.

6.3.7 **return** $\{S\}$.

Even if item j can not be packed in I_1, the new solution (neighbor) may be better than the actual one whether the area of its rectangular closure is smaller than the last. The probability that this type is chosen is q_1;

2. In this case, a neighbor is obtained by changing item $i \in I_1$ with an item $j \in I_2$. The change only occurs if the area of the item j is greater than the area of item i. After the change, if all the items in I_1 can be packed, this new solution (neighbor) is better than the actual one, since the area occupied by this neighbor is greater than the area of actual solution. Its probability is given by q_2;

3. In the last case, a neighbor is obtained by inserting an item $j \in I_2$ into I_1. Now, if all the items in I_1 can be packed, this solution is clearly better than the actual. Its probability is q_3.

Each neighbor type t is selected by the algorithm LocSEARCH at random (line $6.3.3$) in agreement with each probability value. We assume that the probability to type $(1.)$, that is q_1, was set to be greater than the others. For all types t previously commented the insertions and permutations happened only for the last $p_q\%$ items of I_1. With the changes happening at the end of I_1, the packing process can continue from that position.

At line $6.3.5$ all the neighbors were calculated and one with the best-improvement is selected as the actual solution S. Notice that an optimal packing S' is obtained when all the items in I_1 were packed into the bin and I_2 is empty. So, the algorithm LocSEARCH can halt.

At each moment that a neighbor is computed, the set I_1 has a new configuration. Thus, it is first verified if the items in I_1 generate a feasible packing S', and then, S' is compared with the actual solution S.

We present in Algorithm 6.4 the GRASP based heuristic to solve the 2KPI-(0/1) problem.

This heuristic uses, the algorithm IniSOL to generate initial solutions and the algorithm LocSEARCH to improve the initial solutions, if possible.

Algorithm 6.4: GRASP algorithm to solve the 2KPI-(0/1) problem.

Input : set of items I; set of rotations Θ for items in I; dimensions of the bin (L, W); number of iterations $mIter$; number of iterations x; percentage p of items to be chosen at random; probabilities (q_1, q_2, q_3) for each type of neighbors.
Output : solution S.

6.4.1 $S^* \leftarrow \{\ \}$.
6.4.2 **for** $i \leftarrow 1$ **to** $mIter$ **do**
6.4.3 $\quad (S, I_1, I_2) \leftarrow \text{IniSOL}(I, \Theta, L, W, p)$.
6.4.4 $\quad S' \leftarrow \text{LocSEARCH}(S, I_1, I_2, L, W, x, q_1, q_2, q_3)$.
6.4.5 $\quad$ **if** S' *is better than* S^* **then**
6.4.6 $\quad\quad S^* \leftarrow S'$.
6.4.7 **return** (S^*).

The GRASP algorithm repeats this search until a maximum number of iterations is reached. On each iteration an initial solution is generated by algorithm IniSOL (line $6.4.3$) and the improvement step is done by algorithm LocSEARCH (line $6.4.4$). The best obtained solution is stored in S^* and returned at end. If the values of S' and S^* are equal, the algorithm selects one with rectangular closure of smallest area.

6.3 Algorithm for the 2KPI Problem

In this section we present an algorithm to solve the 2KPI problem. This algorithm first packs subsets of irregular items into small rectangles with a high occupancy ratio, and then, it uses the exact dynamic programming algorithm proposed by Cintra *et al.* (2008) [27] to solve the 2KP problem considering these rectangular items. Here, we also assume that the value of each item corresponds to its area.

The algorithm that is used to pack subsets of irregular items into small rectangles is denoted by GenRET and is presented in the Algorithm 6.5. The idea is to pack items one by one and, consequently, to create partial rectangles that grows every time a new item is packed. If the generated rectangle reaches a good occupancy ratio, it is stored with the set of irregular items packed into it.

The algorithm GenRET repeats this search until a maximum number of iterations or rectangles is reached. On each iteration a rectangle rtg is generated from the set I' given that $pu\%$ of the items from I are chosen at random (see lines $6.5.2 - 6.5.4$). The strategy is to use the algorithm PackS to obtain packings with a certain minimum occupation ratio of subsets of items in I' into rectangles rtg (lines $6.5.5 - 6.5.11$).

Algorithm 6.5: GenRET.

Input : set of items I; set of rotations Θ for items in I; minimum occupancy ratio B; maximum allowable dimensions (L_R, W_R) for a rectangle; maximum number m of iterations; maximum number r of rectangles to be generated; percentage pu of items to be chosen at random.

Output : set of rectangles R whose occupancy ratio is at least B.

6.5.1 $R \leftarrow \{\ \}$.
6.5.2 **for** $j, l \leftarrow 1$ **to** $j \leq m$ **AND** $l \leq r$ **do**
6.5.3 $rtg \leftarrow \{\ \}$.
6.5.4 $I' \leftarrow$ choose $pu\%$ of items from I at random.
6.5.5 **foreach** $i \in I'$ **do**
6.5.6 $(flag, x, y, \theta) \leftarrow \text{PackS}(i, \Theta, L_R, W_R, rtg)$.
6.5.7 **if** $flag \neq -1$ **then**
6.5.8 Pack item i into rtg on point (x, y) and at rotation θ.
6.5.9 $ocupp \leftarrow$(sum up of the area of items packed in rtg) / (area of rectangular closure of rtg).
6.5.10 **if** $occup \geq B$ **then**
6.5.11 $R \leftarrow R \cup \{rtg\}$.
6.5.12 **return** $\{R\}$.

It is worth mentioning that algorithm GenRET may not generate a rectangle with occupancy ratio smaller than B. So, we use this algorithm as subroutine in Algorithm 6.6, denominated by PackAllRET. The algorithm PackAllRET first pack items into rectangles with high occupancy ratio. Next, the occupancy ratio is decremented on each iteration by a small value of ϵ so new rectangles may be generated.

Algorithm 6.6: PackAllRET.

Input : set of items I; set of rotations Θ for items in I; initial occupancy ratio B; maximum allowable dimensions (L_R, W_R) for a rectangle; maximum number m of iterations; maximum number r of rectangles to be generated; percentage pu of items to be chosen at random; value ϵ used to decrement the occupancy ratio.

Output : set of rectangles R with different occupancy rates.

6.6.1 $R \leftarrow \{\ \}$.
6.6.2 **while** $B > 0$ **AND** *there are items not packed yet* **do**
6.6.3 $I' \leftarrow$ items of I that are not packed in any rectangle in R.
6.6.4 $R' \leftarrow \text{GenRET}(I', \Theta, B, L_R, W_R, m, r, pu)$.
6.6.5 $R \leftarrow R \cup R'$.
6.6.6 $B \leftarrow B - \epsilon$.
6.6.7 **return** $\{R\}$.

In accordance with algorithm PackAllRET, at line 6.6.3 a set I' of items that are not packed in any rectangle is obtained. This set is used as input to the algorithm GenRET so that a new set of rectangles R' may be obtained with occupancy rates of at least B (line 6.6.4). In this case, the set R' is stored and the value of B is decremented by ϵ. The value of ϵ is chosen to allow all the items in I to be packed.

It is worth mentioning that all the presented algorithms (here and in the previous sections) have polynomial worst-case time and space complexity.

Since the irregular items are packed into rectangles with good occupancy rates, we can use any algorithm developed for the 2KP problem that consider rectangular items. Therefore, we use the exact algorithm developed by Cintra *et al.* (2008) [27], as it is very fast even when solving hard instances. Moreover, it will be used to generate columns in the column generation heuristic presented for the 2CSI problem.

The exact algorithm of [27] consists in solving the recurrence formula of Beasley (1985) [8] considering guillotine cutting patterns generated over the *discretization points* of Herz [48]. We consider the same algorithm of [27], but restricted to patterns generated only over the *reduced raster points* (raster points) of Scheithauer (1997) [83].

A *guillotine cutting pattern* is defined as the pattern obtained by orthogonal guillotine cuts applied to an original bin and to the subsequent smaller bins that were obtained after each cut. A *cutting pattern* (or only *pattern*) consists in each possible way to cut orthogonally the bin and get the final items. The *guillotine cuts* are cuts that goes from one edge of the bin to the opposite one, and are parallel to the remaining edges.

Let (L, W, l, w, v) be a tuple representing an instance for the 2KP problem for rectangular items, where $l = (l_1, \ldots, l_n)$ and w and v are lists defined likewise. The bin has dimensions (L, W) and each item i has length l_i, width w_i and value v_i, for $i = 1, \ldots, n$.

A *discretization point of the length* (resp., of the *width*) is a value $i \leq L$ (resp., $j \leq W$) obtained by a non-negative integer linear combination of $l = (l_1, \ldots, l_n)$ (resp., $w = (w_1, \ldots, w_n)$). We consider the set of all discretization points for length and width represented by P and Q, respectively. Thus, the sets of raster points $\tilde{P}$ (related to P) and $\tilde{Q}$ (related to Q) are calculated by:

$$\begin{array}{ll} \tilde{P} := \{\langle L - r\rangle_P \mid r \in P\}; & \text{where } \langle s\rangle_P = \max\{t \in P \mid t \leq s\}; \\ \tilde{Q} := \{\langle W - u\rangle_Q \mid u \in Q\}; & \text{where } \langle a\rangle_Q = \max\{b \in Q \mid b \leq a\}. \end{array} \tag{6.1}$$

The sets of raster points are computed by the algorithm RRP: first, such algorithm computes the the sets of discretization points by the dynamic programming algorithm DDP (*Discretization using Dynamic Programming*) presented in [27]; next, RRP selects the set of raster points following Eq. (6.1). The algorithm RRP has $O(nD)$ time complexity (pseudo-polynomial) where D assume $\max\{L; W\}$.

Given a rational number $x_r \leq L$ (resp. $y_r \leq W$), we denote by $p(x_r)$ (resp. $q(y_r)$) as the

smallest raster point of length (resp. width) that is greater than or equal to the given input. More formally,

$$\begin{array}{l} p(x_r) = \max\{i|\ i \in \tilde{P},\ i \leq x_r\}; \\ q(y_r) = \max\{j|\ j \in \tilde{Q},\ j \leq y_r\}. \end{array} \tag{6.2}$$

The recurrence formula of Beasley that computes the value $G(l^*, w^*)$ of an optimum guillotine solution for a bin with dimensions (l^*, w^*), and that uses the functions defined in Eq. (6.2) is described next:

$$G(l^*, w^*) = \max \left\{ \begin{array}{l} g(l^*, w^*); \\ \max(\{G(l', w^*) + G(p(l^* - l'), w^*)|\ \ l' \in \tilde{P},\ \ l' \leq l^*/2\}); \\ \max(\{G(l^*, w') + G(l^*, q(w^* - w'))|\ \ w' \in \tilde{Q},\ \ w' \leq w^*/2\}); \end{array} \right\}, \tag{6.3}$$

where $g(l^*, w^*)$ denotes the value of the most valuable item that can be cut in a bin with dimensions (l^*, w^*). This value is 0, if no item can be cut in such rectangle.

We denote by DP2KPG the dynamic programming algorithm that solves the Eq. (6.3) using the raster points for an instance $I = (L, W, l, w, v)$ of the 2KP problem with rectangular items. According to [27], the worst-case time complexity of the algorithm DP2KPG is $O(nL + nW + |\tilde{P}|^2|\tilde{Q}| + |\tilde{P}||\tilde{Q}|^2)$. The required space complexity is $O(L + W + |\tilde{P}||\tilde{Q}|)$.

Now, we can describe the algorithm that solves the 2KPI problem, which we denote by Solve2KPI:

1. Generate the set R of rectangular items by algorithm PackAllRET;
2. Create an instance I for the 2KP problem for rectangular items by considering the rectangles in R, so for each rectangle $r \in R$ its values v_r is given by the total sum of the item values packed in it;
3. Execute algorithm DP2KPG for instance I where the set of raster points was first computed by RRP;
4. Return the solution computed by DP2KPG as the final solution for the 2KPI problem.

Note that in the final solution each rectangle represents a subset of items (polygons) packed in it.

6.4 The Column Generation Heuristic for the 2CSI Problem

In this section, we consider the Two-dimensional Cutting Stock problem with irregular items, denoted by 2CSI. We use a column generation algorithm to solve such problem where the

columns are generated by algorithm Solve2KPI.

The integer linear formulation for the Cutting Stock problem based on cutting patterns is very known. Let $\mathcal{P}$ be a set of cutting patterns in which $|\mathcal{P}| = m$ denote its size. The linear formulation uses a matrix P with order $n \times m$ where each column represent a cutting pattern and a vector d of demand of the items. Each column has the number of copies of each item in the corresponding pattern. We use the integer variable x_j for each pattern $j \in \mathcal{P}$, each one indicates how many times the pattern j is used in the solution. Thus, the following linear program is a relaxation of the equivalent integer formulation:

$$\begin{array}{lllll} minimize & \displaystyle\sum_{j \in \mathcal{P}} x_j & & & \\ subject\ to & Px & \geq & d & \\ & x_j & \geq & 0 & \forall\, j \in \mathcal{P}. \end{array} \tag{6.4}$$

We use the algorithms presented in Cintra *et al.* (2008) [27] to solve the 2CSI problem. The main idea of such algorithms are to solve the linear formulation (6.4) using the column generation approach proposed by Gilmore and Gomory (1961; 1963) [41, 42]; to consider the integer part of the solution; and, to deal with the residual problems in an iterative way, that is using the same approach combined with a primal heuristic. The primal heuristic returns cutting patterns that causes a perturbation of those residual problems.

Notice that in each iteration, the algorithm must generate a new column to be inserted into the basis. To this end, the dual value y_i is computed for each item i. The new column (pattern) must satisfy that $\sum_{i=1}^{n} y_i z_i > 1$, where z_i is the number of times that item i appear in such pattern. Of course we can use any algorithm developed for the 2KPI problem to compute these patterns.

The algorithm used to solve the linear programming (6.4) is the algorithm SimplexCG$_2$ shown in Cintra *et al.* (2008) [27]. It corresponds to the algorithm Simplex with a subroutine to generate columns which is discussed next. More details about this algorithm can be found in [25]. Our implementation consider the matrix $I_{n \times n}$ as the identity matrix with n patterns, each one with items of one shape and one orientation.

The subroutine used to generate the columns uses the algorithm Solve2KPI with a simple modification. Remember that algorithm Solve2KPI packs the items observing their areas. However, now each item has a value that do not necessary corresponds to its area (value of the dual). The simple modification consists in to generate rectangles with occupancy ratio given by B times C, where B is the minimum occupancy ratio previously defined and C is computed on each iteration of PackAllRET, and it is returned by Algorithm 6.7, so called by computeC. So, we consider $B \times C$ as input for GenRET in algorithm PackAllRET (see line 6.6.4).

In algorithm computeC the values of area a_i and v_i for each item i are not necessarily equal. It is also necessary to modify the way that the variable *ocupp* is calculated in algorithm GenRET

Algorithm 6.7: computeC.

Input : set of items I each one with area a_i and value v_i.
Output: upper bound C.

6.7.1 Find $i \in I$ such that the value of $\frac{v_i}{a_i}$ is maximum.
6.7.2 $C \leftarrow \frac{v_i}{a_i}$.
6.7.3 **return** $\{C\}$.

(see line 6.5.9). Now, $occup$ is given by: (total sum of the values of the items packed in rtg) / (area of the rectangular closure of rtg).

The algorithm SimplexCG$_2$ may return a fractional solution, then we also extend the algorithm CG^p (presented in [27]) that receives the solution returned by SimplexCG$_2$, and returns an integer solution for the 2CSI problem. The algorithm CG^p uses as subroutine a primal heuristic to obtain cutting patterns that cause a perturbation of the residual problems. We denominate the algorithm that solves the 2CSI problem by Solve2CSI.

The primal heuristic considered corresponds to the algorithm M-HFF, that is a modified version of the heuristic HFF (*Hybrid First Fit*) to deal with demands. As the heuristic M-HFF can not handle with irregular items, we consider the smallest rectangle where each irregular item can be packed as input for such heuristic. Details about HFF are shown in [24].

The algorithm CG^p has exponential time complexity, but as mentioned by Cintra *et al.* (2008) this algorithm in fact halts, because after a finite number of iterations the demand will be fulfilled.

6.5 Computational Results

All the algorithms above mentioned were implemented in $C/C++$ language and the geometry library used was the CGAL (Computational Geometry Algorithms Library) available at the url: `http://www.cgal.org/`.

The tests were performed on computer with processor Intel® Core^{TM} 2 Quad 2.4 GHz, 4 GB of RAM memory and *Linux* operating system. The linear systems in the column generation algorithms were solved by COIN-OR CLP solver [28].

The instances used were adapted from the Two-Dimensional Irregular Strip Packing problem and they can be found at the url: `http://ww.fe.up.pt/esicup/`. In these instances the following informations are available: the quantity of items where each item is represented by a polygon; the set of allowable rotations for these items; and, the length of the strip. The objective of such problem is to pack all the items into the strip so that the height of the used part of the strip is minimized.

Anyway for our problems the bins have fixed rectangular dimensions, so that the length of

the bin is the same length used for the strip and the width of the bin consists at height presented in Gomes and Oliveira (2006) [45]. For the 2KPI and 2CSI problems only the integer part of the dimensions of the bin were considered. Besides that, it was necessary to generate a value of demand for the instances of the 2CSI problem. Thus, we generate an integer value of demand for each item. Such value was randomly chosen in the interval $[1,\ 100]$.

For future researches as well as benchmarks purposes such instances are available at the url: `http://www.loco.ic.unicamp.br/irregular2d/`.

Tables 6.1 presents some initial informations about the instances used by the algorithms. A total of 15 instances were considered for numerical tests. Each row of this table has the following information: instance name (Name); quantity of items (n); length of the bin (L); width of the bin (W); and, set of allowable rotations for the items (Rotations);

Table 6.1: Informations about the instances under consideration.

Name	n	L	W	Rotations
FU	12	38	34	(0, 90, 180)
JACKOBS1	25	40	13	(0, 90, 180)
JACKOBS2	25	70	28.2	(0, 90, 180)
SHAPES0	43	40	63	(0)
SHAPES1	43	40	59	(0, 180)
SHAPES2	28	15	27.3	(0, 180)
DIGHE1	16	100	138.13	(0)
DIGHE2	10	100	134.05	(0)
ALBANO	24	4900	10122.63	(0, 180)
DAGLI	30	60	65.6	(0, 180)
MAO	20	2550	2058.6	(0, 90, 180)
MARQUES	24	104	83.6	(0, 90, 180)
SHIRTS	99	40	63.13	(0, 180)
SWIM	48	5752	6568	(0, 180)
TROUSERS	64	79	245.75	(0, 180)

The GRASP based heuristic as well as all the algorithms used as subroutines depend on several parameters which need to be well defined, since the quality of the solution depends on such parameters. Likewise, the algorithms Solve2KPI and Solve2CSI (and their subroutines) depends of various parameters too.

In Table 6.2 we summarize the values for all those parameters considering all the algorithms previously discussed. It is worth mentioning that such values were chosen after an extensive execution of tests considering the instances presented in Table 6.1. Each parameter was tested individually and also together with others in order to set a final value for it.

Table 6.3 illustrates the results obtained by the GRASP based heuristic for the 2KPI-(0/1) problem. The following information are shown in each row of this table: instance name (Name); the time spent (in seconds) to compute all the no-fit polygons (Time of NFP); the quantity of

Table 6.2: Parameters required by algorithms previously discussed.

Parameter	Value considered	Main algorithm that uses such parameter
p	10%	GRASP
x	25	GRASP
(q_1, q_2, q_3)	(70, 20, 10)	GRASP
p_q	60%	LocSEARCH
$mIte$	15	GRASP
pu	80%	Solve2KPI, Solve2CSI
B	90%	Solve2KPI, Solve2CSI
m	15	Solve2KPI, Solve2CSI
r	two times the quantity of items n	Solve2KPI, Solve2CSI
ϵ	10%	Solve2KPI, Solve2CSI
(L_R, W_R)	same values considered for (L, W)	Solve2KPI, Solve2CSI

items in the solution (Quantity of Items); the area of the bin occupied by items (Area Occupied); the time spent (in seconds) to solve the respective instance (Time of GRASP); and, the total time spent: Time of NFP + Time of GRASP.

Table 6.3: Performance of the GRASP based heuristic for the 2KPI-(0/1) problem.

Name	Time of NFP (s)	Quantity of Items	Area Occupied	Time of GRASP (s)	Total Time (s)
FU	0.26	12	0.8382	21.79	22.05
JACKOBS1	59.49	25	0.7538	8.30	67.79
JACKOBS2	53.80	25	0.6844	565.71	619.51
SHAPES0	51.32	41	0.6016	1552.46	1603.78
SHAPES1	202.56	41	0.6424	3891.60	4094.16
SHAPES2	17.90	26	0.7289	1048.12	1066.02
DIGHE1	0.12	16	0.7240	10.68	10.80
DIGHE2	0.15	10	0.7460	0.07	0.22
ALBANO	14.40	23	0.8038	961.59	975.99
DAGLI	26.16	29	0.7586	1106.40	1132.56
MAO	102.91	20	0.7160	120.42	223.33
MARQUES	57.50	24	0.8274	217.77	275.27
SHIRTS	208.49	96	0.7702	14317.13	14525.62
SWIM	1088.21	46	0.6427	39781.43	40869.64
TROUSERS	48.22	62	0.7866	5796.59	5844.81

Observing Table 6.3 we note that the solution of the following instances could not pack all the items: SHAPES0, SHAPES1, SHAPES2, ALBANO, DAGLI, SHIRTS, SWIM and TROUSERS. It corresponds to 8 out of 15 instances. On the other hand, the heuristic found optimal solutions for the others 7 instances. The percentage of the area of the bin that was occupied was of 73.5%, on average, and at worst case was of 60.16%, for instance SHAPES0. The time spent to compute the no-fit polygons was smaller compared to that required by the heuris-

tic. On average, the calculus of the NFP required 128.77 seconds, against 4626.67 seconds of the last one.

Anyway, the results in Table 6.3 demonstrates good occupancy rates, since the considered instances have a large number of items completely irregular and they are considered hard to solve. Moreover, better results may be found if we do not consider conservative values for the parameters of the GRASP heuristic. However, the time required to solve the instances may increase accordingly to.

Table 6.4 presents the results of the numerical experiments for the 2KPI problem. For each instance the algorithm Solve2KPI was executed 10 times, and the average runtime and area of occupation of the bin were returned as final solution.

In each line of Table 6.4 we have: instance name (Name); the best occupation of the bin (Best Occupation); the average value of occupation (Avg. Occupation); the average time (of CPU) in seconds spent by algorithm Solve2KPI to solve the respective instance (Avg. Time); and, the time spent (in seconds) to compute the no-fit polygons (Time of NFP).

Table 6.4: Performance of the algorithm Solve2KPI for the 2KPI problem.

Name	Best Occupation	Avg. Occupation	Avg. Time (s)	Time of NFP (s)
FU	0.9892	0.9892	5.01	0.26
JACKOBS1	1.0000	0.9870	49.60	59.49
JACKOBS2	1.0000	0.9851	66.08	53.80
SHAPES0	0.6190	0.5873	134.43	51.32
SHAPES1	0.6949	0.6893	248.15	202.56
SHAPES2	0.9284	0.9183	49.37	10.48
DIGHE1	0.7631	0.6909	7.62	0.12
DIGHE2	0.7791	0.7657	3.14	0.15
ALBANO	0.9653	0.9653	37.93	14.40
DAGLI	0.9256	0.9196	50.99	26.16
MAO	0.9812	0.9644	71.53	102.91
MARQUES	0.9606	0.9515	43.76	57.50
SHIRTS	1.0000	1.0000	508.92	208.49
SWIM	0.7590	0.7272	2206.42	1088.21
TROUSERS	1.0000	0.9986	193.54	48.22

As observed in Table 6.4, the instances where there are several rectangular items or few items completely irregular allow the algorithm PackAllRET to generate rectangles with high occupancy rates. The instances with occupancy rates greater than 90% were: FU, JACKOBS1, JACKOBS2, SHAPES2, ALBANO, DAGLI, MAO, MARQUES, SHIRTS and TROUSERS, that corresponds to 66.67% of the instances. On the other hand, the instances SHAPES0, SHAPES1, DIGHE1, DIGHE2 and SWIM have several items completely irregulars, so that the average occupancy ratio decrease accordingly to. On average, the average occupancy ratio and the average time plus the time of NFP were of 87.60% and 373.37 seconds, respectively.

Table 6.5: Results obtained for the 2CSI problem.

Name	Solution	LB	Difference (%)	Time (s)	Columns
FU	81	81.0	0.000%	70.91	168
JACKOBS1	52	51.0	1.961%	2112.33	465
JACKOBS2	47	47.0	0.000%	2576.83	606
SHAPES0	38	36.0	5.556%	15409.90	2260
SHAPES1	32	31.0	3.226%	68585.42	3089
SHAPES2	35	33.0	6.060%	6260.68	1176
DIGHE1	61	61.0	0.000%	103.50	287
DIGHE2	48	48.0	0.000%	16.01	103
ALBANO	35	33.0	6.060%	3443.47	862
DAGLI	30	28.0	7.142%	8427.17	1517
MAO	47	47.0	0.000%	2337.35	462
MARQUES	51	51.0	0.000%	3983.01	815
SHIRTS	17	16.0	6.250%	343859.61	8325
SWIM	22	21.0	4.761%	11208.77	1652
trousers	20	19.0	5.263%	171122.16	6859

The results returned by algorithm Solve2CSI for the 2CSI problem are shown in Table 6.5. Each column of this table presents: instance name (Name); solution computed by algorithm Solve2CSI (Solution); the lower bound (LB) for the value of an optimum integer solution that is given by solving the linear relaxation (6.4) by algorithm Solve2CSI; the difference (in percentage) on number of bins computed by Solve2CSI and LB; the time in seconds spent by Solve2CSI (Time); the number of columns generated (Columns).

Observing Table 6.5 we can note that the solution obtained using algorithm Solve2CSI is equal to the corresponding lower bound (LB) for instances: FU, JACKOBS2, DIGHE1, DIGHE2, MAO and MARQUES. It corresponds to 40.0% of the instances under consideration. For the other instances the difference was of 5.142%, on average, and, at worst case was of 7.142%, for instance DAGLI. However, if we consider the number of bins required such difference was of at most two bins (see instances: SHAPES0, ALBANO and DAGLI).

Some solutions were returned after 343859.42 seconds (≈ 96 hours) of CPU processing, at worst case. The CPU time spent on average was of 42634.47 seconds, as presented in Table 6.5. The number of columns generated was of 1909.73, on average. It shows that the column generation heuristic is useful to deal with this variant of the Cutting Stock problem, that is the case with irregular items. However, it requires high CPU time for solving instances that have several items completely irregulars.

All the results here were obtained for those parameters considered in Table 6.2. These results were obtained after several executions of the algorithms discussed in this paper, so they are average values. Naturally, better results may be obtained if we change the value of those parameters in Table 6.2. However, if better results are required as to increase the occupancy

ratio of the bins, the time required to solve each instance may also increase.

6.6 Concluding Remarks

In this paper we presented heuristics to deal with the following problems on two-dimensional version: 0/1 Knapsack, Unconstrained Knapsack and Cutting Stock. For all these problems we consider the case where the items are represented by polygons, that is irregular shapes (with holes).

To handle the geometry we codify the robust no-fit polygons generator proposed by Burke *et al.* (2007) [19], and an extended version of the search algorithm developed by Adamowicz and Albano (1976) [1] to deal with the arrangement of the items inside the bin.

For the Irregular 0/1 Knapsack problem we presented a GRASP based heuristic that shows to be efficient in the cases where there are few irregular items in the instance, due to the high occupancy ratio B that we set for the bin on numerical tests. Optimal solutions were obtained for 7 out of 15 instances within a reasonable CPU time. Better results may be obtained for the cases where conservative values of the parameters are not considered, however the required CPU time may also increase.

For the Irregular Unconstrained Knapsack problem, the proposed approach obtained for 66.67% of the instances solutions that occupies 90% of the bin area spending little CPU time.

For the Irregular Cutting Stock problem the column generation heuristic showed to be efficient not only for the case with rectangular items (as presented by [27]) but also for the case with irregular items. Solutions equal to the lower bound were computed for 6 out of 15 instances. For the other cases, the difference between the solution computed and the lower bound was by at most two bins. As mentioned, the only drawback was the required CPU time, that can be considered high for instances of moderate size.

After all, we can state that the proposed algorithms are suitable and well-defined to solve practical instances of moderate size. It is worth mentioning that optimum or quasi-optimum solutions were obtained in a satisfactory amount of computational time.

Chapter 7

Conclusões

Nesta tese trabalhamos com as versões bi- e tridimensionais dos seguintes problemas de corte e empacotamento: mochila, corte de estoque e empacotamento em faixa. Variantes destes problemas, consideradas $\mathcal{NP}$-difíceis, foram investigadas e, para cada uma delas, propomos abordagens exatas e/ou heurísticas.

O Capítulo 2 implementa e analisa soluções para as versões tridimensionais guilhotinada e com k estágios de corte dos problemas da mochila irrestrita, do corte de estoque, do corte de estoque com contêineres de tamanhos variados e do empacotamento em faixa. As variantes do problema da mochila irrestrita são resolvidas via programação dinâmica, enquanto que os demais problemas e suas variantes são resolvidos via geração de colunas, sendo que a cada iteração deste método, um novo padrão é definido via solução da correspondente versão do problema da mochila irrestrita. Os diversos testes realizados indicam que estes algoritmos são úteis para resolver problemas reais de pequeno a médio porte. Como trabalho futuro seria interessante tentar estender os algoritmos de programação dinâmica do problema da mochila irrestrita para a sua versão restrita. Também, estudar variantes dos problemas em foco em que não necessariamente é preciso empacotar todos os itens da instância da entrada. Isto tem aplicação direta, por exemplo, no transporte aéreo, visto que é preciso decidir sempre que um voo chega/parte quais objetos devem ser levados ou deixados para um próximo voo.

No Capítulo 3 estudamos os problemas da mochila irrestrita e do corte de estoque nas suas versões bidimensionais considerando cortes não-guilhotinados. Investigamos as variante em que os itens possuem orientação fixa e aquela em que os itens podem ser rotacionados ortogonalmente. Neste capítulo mostramos empiricamente que os algoritmos propostos para uma versão particular do problema da mochila irrestrita, que permite a rotação dos itens, são úteis neste contexto, isto é, para resolver o caso mais geral. Ainda no contexto do problema da mochila irrestrita, apresentamos uma instância na qual a Abordagem L falha em computar a solução ótima, introduzimos a Abordagem $L^{(k)}$, que reduz o esforço computacional requerido pela Abordagem L, e provamos a questão em aberto relacionada a usar somente os *reduced*

raster points na Abordagem L. No caso do problema do corte de estoque, uma heurística baseada em geração de colunas, que faz uso dos algoritmos previamente desenvolvidos para o problema da mochila irrestrita foi apresentada. Nos testes computacionais realizados obtivemos soluções melhores que as previamente existentes para grande parte das instâncias utilizadas. Um possível trabalho futuro seria a extensão destes algoritmos com o intuito de resolver a versão tridimensional do problema da mochila irrestrita para cortes não-guilhotinados. Também, seria interessante investigar uma abordagem similar a L em que os cortes formariam peças retangulares, na forma de L e/ou na forma de U. Então, tentar provar se esta nova abordagem forneceria (ou não) a solução ótima para o problema em questão.

O artigo apresentado no Capítulo 4 propõe heurísticas que geram empacotamentos em níveis, usam a ideia de pontos de canto e um algoritmo *branch-and-cut* para resolver a versão bidimensional do problema de empacotamento em faixa conservando a estabilidade (condições de equilíbrio estático para corpos rígidos) e respeitando uma certa ordem de descarregamento. Para este problema foi apresentado um modelo de programação inteira. O algoritmo *branch-and-cut* depende da discretização do recipiente, sendo a complexidade do problema intimamente relacionada à quantidade de pontos em tal discretização. Fizemos uma análise exata das condições de equilíbrio estático e propusemos um algoritmo, usado como planos de corte, para determinar se um dado empacotamento bidimensional é estável. Nos resultados computacionais as heurísticas se mostraram bastante eficazes para resolver o problema em foco. Elas retornaram soluções cuja altura é ótima para diversas instâncias em pouco tempo computacional. Um trabalho futuro interessante seria estender a metodologia proposta para a estabilidade (estática) para também lidar, de forma exata, com a estabilidade dinâmica. Então, considerar situações em que há variação de velocidade, aceleração e/ou inclinação dos itens/recipiente. Outra extensão deste metodologia seria lidar com o caso de empacotamentos tridimensionais, também de forma exata. Para o modelo de programação linear, seria interessante investigar restrições que permitem reduzir simetrias no padrão de empacotamento. Também, se é possível utilizar, sem perda de generalidade, uma malha de pontos representada por conjuntos de menor cardinalidade, como os *reduced raster points*.

Uma extensão da formulação inteira apresentada no Capítulo 4 é feita no Capítulo 5 para tratar da versão bidimensional do problema de empacotamento em faixa sujeito a restrição de ordem e ao balanceamento de carga. Como resultado, dois modelos de programação inteira são apresentados: o primeiro possui uma malha de pontos mais refinada, enquanto que o segundo possui uma malha com menos pontos. Estes modelos foram resolvidos por um resolvedor comercial considerando instâncias geradas de forma randômica. Bons resultados foram obtidos quando os dois modelos são combinados, isto é, a solução gerada por um modelo é usada como solução inicial para o outro modelo. Deste modo, conseguimos resolver a otimalidade grande parte das instâncias considerando o tempo máximo de processamento imposto. Assim como nos resultados obtidos no Capítulo 4, estes modelos são fortemente dependentes da malha de pontos

utilizada, de modo que quanto mais refinada for esta malha, maior será o consumo de tempo computacional. Uma extensão natural dos modelos propostos seria lidar com outras restrições práticas, como considerar o empacotamento completo de grupos de itens, assumir prioridade para os itens, manter itens separados dentro do recipiente e questões como fragilidade, as quais envolve um limite quanto ao número de itens (peso) que outro item suportaria. Poderia também ser investigado com mais detalhes os parâmetros adotados pelo resolvedor comercial. Todos os testes foram realizados para os valores padrão do resolvedor. Outro ponto a ser investigado consiste no desenvolvimento de heurísticas que gerem soluções de qualidade para o problema em foco. A primeiro momento, a restrição de balanceamento de carga não se demonstrou de fácil consideração para gerar uma heurística de qualidade.

No Capítulo 6 estudamos o caso em que os itens são representados por polígonos convexos e não-convexos dos problemas da mochila 0/1, da mochila irrestrita e do corte de estoque nas suas versões bidimensionais. Fizemos uso do conceito de *No-Fit polygons* para lidar com a geometria dos itens e, então, obter pontos viáveis para empacotá-los. Os pontos escolhidos foram os que minimizavam a área da envoltória retangular e convexa do empacotamento. Para a versão do problema da mochila 0/1, propomos uma heurística baseada em GRASP. Em se tratando da versão do problema da mochila irrestrita, desenvolvemos uma rotina que empacota itens irregulares em retângulos considerando uma alta taxa de ocupação. Estes retângulos foram usados para criar uma instância do problema da mochila irrestrita que lida com itens retangulares, a qual foi resolvida por um algoritmo exato de programação dinâmica. Por fim, a versão do problema do corte de estoque usou da mesma heurística de geração de colunas discutida nos capítulos anteriores. Antes de chamar o algoritmo desenvolvido para o problema da mochila irrestrita, que trata de itens irregulares, para gerar uma nova coluna, retângulos compostos de itens irregulares eram gerados observando o valor das variáveis duais e da área dos itens. Os resultados obtidos foram satisfatórios, visto que conseguimos para grande parte das instâncias soluções com alta taxa de ocupação do recipiente. Porém, o tempo computacional requerido foi elevado para o problema do corte de estoque. Alguns pontos promissores que podem ser investigados incluem implementar novos algoritmos para lidar com a geometria dos itens, bem como implementar uma biblioteca geométrica que otimize as operações mais utilizadas, como as operações que envolvem o cálculo de interseções entre os itens. No caso do ponto escolhido para empacotar um item, além de escolher o ponto que minimize a área da envoltória retangular e convexa, também poderia se considerar o ponto que minimize a área do menor polígono que envolve a união dos itens. Parece promissor estudar e propor modelos de programação linear que lidem de forma eficiente com itens irregulares, visto que grande parte das estratégias encontradas na literatura consideram heurísticas.

Trabalhos futuros poderiam abordar o desenvolvimento de algoritmos de aproximação para os problemas relatados nesta tese. Um estudo comparativo envolvendo tais tipos de algoritmos com os desenvolvidos nesta tese pode ser interessante tanto do ponto de vista teórico (desen-

volver os algoritmos de aproximação) como prático (implementar e compará-los com outros algoritmos).

Por fim, podemos afirmar que esta tese tem sua contribuição no campo da ciência tanto em termos quantitativos (número de problemas abordados e soluções propostas), como em termos qualitativos (resultados originais e de forte aplicação prática).

Bibliography

[1] M. Adamowicz and A. Albano. Nesting two-dimensional shapes in rectangular modules. *Computer-Aided Design*, 8:27–33, 1976.

[2] A. Albano and G. Sapuppo. Optimal allocation of two-dimensional irregular shapes using heuristic search methods. *IEEE Trans. Syste., Man, Cybern., SMC-10*, 5:242–248, 1980.

[3] R. Alvarez-Valdes, F. Parre no, and J. M. Tamarit. Reactive grasp for the strip-packing problem. *Computers & Operations Research*, 35:1065–1083, 2008.

[4] R. Alvarez-Valdes, A. Parajon, and J. M. Tamarit. A computational study of lp-based heuristic algorithms for two-dimensional guillotine cutting stock problems. *OR Spectrum*, 24(2):179–192, 2002.

[5] C. Alves and J. M. Valério de Carvalho. A stabilized branch-and-price-and-cut algorithm for the multiple length cutting stock problem. *Computers & Operations Research*, 35:1315–1328, 2008.

[6] M. Arenales and R. Morábito. An and/or-graph approach to the solution of two-dimensional non-guillotine cutting problems. *European J. Operational Research*, 84:599–617, 1995.

[7] M. Bazaraa, J. J. Jarvis, and H. D. Sherali. *Linear Programming and Networks Flows*. John Wiley and Sons, 2003.

[8] J. E. Beasley. Algorithms for unconstrained two-dimensional guillotine cutting. *Journal of the Operational Research Society*, 36(4):297–306, 1985.

[9] J. E. Beasley. An exact two-dimensional non-guillotine cutting tree search procedure. *Operations Research*, 33:49–64, 1985.

[10] J. E. Beasley. OR-Library: distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072, 1990.

[11] J. E. Beasley. A population heuristic for constrained two-dimensional non-guillotine cutting. *European Journal of Operational Research*, 156:601–627, 2004.

[12] F. P. Beer, E. R. Johnston Jr., J. DeWolf, and D. Mazurek. *Mechanics of Materials*. McGraw-Hill Science, 5 edition, 2008.

[13] G. Belov and G. Scheithauer. A branch-and-cut-and-price algorithm for one-dimensional stock cutting and two-dimensional two-stage cutting. *European J. Operational Research*, 171:85–106, 2006.

[14] J. A. Bennell, K. A. Dowsland, and J. F. W. B. Dowsland. The irregular cutting-stock problem - a new procedure for deriving the no-fit polygon. *Journal of the Operational Research Society*, 28:271–287, 2001.

[15] J. A. Bennell and J. F. Oliveira. The geometry of nesting problems: A tutorial. *Journal of the Operational Research Society*, 184:397–415, 2008.

[16] J. A. Bennell and J. F. Oliveira. A tutorial in irregular shape packing problems. *Journal of the Operational Research Society*, 60:93–105, 2009.

[17] E. G. Birgin, R. D. Lobato, and R. Morábito. An effective recursive partitioning approach for the packing of identical rectangles in a rectangle. *Journal of the Operational Research Society*, 61(2):306–320, 2010.

[18] E. E. Bischoff and M. S. W. Ratcliff. Issues in the development of approaches to container loading. *OMEGA*, 23(4):377–390, 1995.

[19] E. K. Burke, R. Hellier, G. Kendall, and G. Whitwell. Complete and robust no-fit polygon generation for the irregular stock cutting problem. *Journal of the Operational Research Society*, 179:27–49, 2007.

[20] A. Caprara and M. Monaci. On the two-dimensional knapsack problem. *Operations Research Letters*, 32:5–14, 2004.

[21] J. N. Cernica. *Strength of Materials*. Holt, Rinehart and Winston, inc., 1966.

[22] S. G. Christensen and D. M. Rousøe. Container loading with multi-drop constraints. *International Transactions in Operational Research*, 16(6):727–743, 2009.

[23] N. Christofides and C. Whitlock. An algorithm for two dimensional cutting problems. *Operations Research*, 25:30–44, 1977.

[24] F. R. K. Chung, M. R. Garey, and D. S. Johnson. On packing two-dimensional bins. *SIAM Journal on Algebraic and Discrete Methods*, 3:66–76, 1982.

[25] V. Chvátal. *Linear Programming*. W. H. Freeman and Company, New York, 1980.

[26] G. Cintra, F. K. Miyazawa, Y. Wakabayashi, and E. C. Xavier. A note on the approximability of cutting stock problems. *European Journal on Operations Research (Elsevier Science)*, 34:2589–2603, 2007.

[27] G. F. Cintra, F. K. Miyazawa, Y. Wakabayashi, and E. C. Xavier. Algorithms for two-dimensional cutting stock and strip packing problems using dynamic programming and column generation. *European Journal of Operational Research*, 191:59–83, 2008.

[28] Coin-OR CLP. Linear programming solver: An open source code for solving linear programming problems. `http://www.coin-or.org/Clp/index.html`.

[29] J. Csirik and A. van Vliet. An on-line algorithm for multidimensional bin packing. *Operations Research Letters*, 13:149–158, 1993.

[30] K. M. Daniels and V. J. Milenkovic. Multiple translational containment. part 1: An approximation algorithm. *Algorithmica*, 19 (Special Issue on Computational Geometry in Manufacturing):148–182, 1997.

[31] G. Dantzig and P. Wolfe. Decomposition principle for linear programs. *Operations Research*, 8:101–111, 1960.

[32] F. Diedrich, R. Thöle R. Harren, K. Jansen, and H. Thomas. Approximation algorithms for 3D orthogonal knapsack. *Journal of Computer Science and Technology*, 23(5):749–762, 2008.

[33] H. Dyckhoff. A typology of cutting and packing problems. *European J. Operational Research*, 44:145–159, 1990.

[34] J. Egeblad, B. K. Nielsen, and A. Odgaard. Fast neighborhood search for two- and three-dimensional nesting problems. *European Journal of Operational Research*, 183:1249–1266, 2007.

[35] D. Fayard, M. Hifi, and V. Zissimopoulos. An efficient approach for large-scale two-dimensional guillotine cutting stock problems. *Journal of the Operational Research Society*, 49(12):1270–1277, 1998.

[36] S. P. Fekete and J. Schepers. A combinatorial caracterization of higher-dimensional orthogonal packing. *Mathematics of Operations Research*, 29:353–368, 2004.

[37] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$-Completeness*. Freeman, San Francisco, 1979.

[38] H. P. Gavin. The three-moment equation for continuous beam. Technical Report, Department of Civil and Environmental Engineering - Duke Unversity, 2009.

[39] M. Gendreau, M. Iori, G. Laporte, and S. Martello. A tabu search algorithm for a routing and container loading problem. *Transportation Science*, 40(3):342–350, 2006.

[40] M. Gendreau, M. Iori, G. Laporte, and S. Martello. A tabu search heuristic for the vehicle routing problem with two-dimensional loading constraints. *Networks*, 51:14–18, 2008.

[41] P. Gilmore and R. Gomory. A linear programming approach to the cutting stock problem. *Operations Research*, 9:849–859, 1961.

[42] P. Gilmore and R. Gomory. A linear programming approach to the cutting stock problem - part II. *Operations Research*, 11:863–888, 1963.

[43] P. Gilmore and R. Gomory. Multistage cutting stock problems of two and more dimensions. *Operations Research*, 13:94–120, 1965.

[44] A. M. Gomes and J. F. Oliveira. A 2-exchange heuristic for nesting problems. *European Journal of Operational Research*, 141:359–370, 2002.

[45] A. M. Gomes and J. F. Oliveira. Solving irregular strip packing problems by hybridising simulated annealing and linear programming. *Journal of the Operational Research Society*, 171:811–829, 2006.

[46] E. Hadjiconstantinou and M. Iori. A hybrid genetic algorithm for the two-dimensional single large object placement problem. *European Journal of Operational Research*, 183:1150–1166, 2007.

[47] D. Halliday, R. Resnick, and J. Walker. *Fundamentals of Physics*. John Wiley & Sons, 8 edition, 2007.

[48] J. C. Herz. A recursive computational procedure for two-dimensional stock-cutting. *IBM Journal of Research Development*, pages 462–469, 1972.

[49] R. C. Hibbeler. *Statics and Mechanics of Materials*. Prentice Hall, 3 edition, 2010.

[50] M. Hifi. Exact algorithms for the guillotine strip cutting/packing problem. *Computers & Operations Research*, 25:925–940, 1998.

[51] E. Hopper and B. C. H. Turton. A review of the application of meta-heuristic algorithms to 2d strip packing problems. *Artificial Intelligence Review*, 16(4):257–300, 2001.

[52] IBM. *ILOG® CPLEX® V12.1 - User's Manual for CPLEX®*. IBM Corporation, 2009.

[53] S. Imahori, M. Yagiura, and T. Ibaraki. Improved local search algorithms for the rectangle packing problem with general spatial costs. *European Journal of Operational Research*, 167:48–67, 2005.

[54] A. Imai, K. Sasaki, E. Nishimura, and S. Papadimitriou. Multi-objective simultaneous stowage and load planning for a container ship with container rehandle in yard stacks. *European Journal of Operational Research*, 171(2):373–389, 2006.

[55] M. Iori, J. Salazar-González, and D. Vigo. An exact approach for the vehicle routing problem with two-dimensional loading constrains. *Transportation Science*, 41(2):253–264, 2007.

[56] K. Jansen and R. Solis-Oba. An asymptotic approximation algorithm for 3D-strip packing. In *Proc. of the 17th annual ACM-SIAM Symposium on Discrete Algorithms*, pages 143–152, 2006.

[57] K. Jansen and R. van Stee. On strip packing with rotations. In *Proc. of the 37th ACM Symposium on Theory of Computing*, 2005.

[58] L. Junqueira, R. Morábito, and D. S. Yamashita. Three-dimensional container loading models with cargo stability and load bearing constraints. *Computers & Operations Research*, doi:10.1016/j.cor.2010.07.017, 2010.

[59] B. L. Kaluzny and R. H. A. David Shaw. Optimal aircraft load balancing. *International Transactions in Operational Research*, 16(6):767–787, 2009.

[60] M. Kenmochi, T. Imamichi, K. Nonobe, M. Yagiura, and H. Nagamochi. Exact algorithms for the two-dimensional strip packing problem with and without rotations. *European Journal of Operational Research*, 198:73–83, 2009.

[61] C. Kenyon and E. Rémila. A near-optimal solution to a two-dimensional cutting stock problem. *Mathematics of Operations Research*, 25:645–656, 2000.

[62] G. F. King and C. F. Mast. Excess travel: causes, extent and consequences. *Transportation Research Record*, 1111:126–134, 1997.

[63] W. C. Lee, H. Ma, and B. H. Cheng. A heuristic for nesting problems of irregular shapes. *Computer Aided Design*, 40:625–633, 2008.

[64] N. Lesh, J. Marks, and A. Mc. Mahon. Exhaustive approaches to 2d rectangular perfect packings. *Information Processing Letters*, 90:7–14, 2004.

[65] L. Lins, S. Lins, and R. Morábito. An l-approach for packing (l, w)-rectangles into rectangular and l-shaped pieces. *Journal of the Operational Research Society*, 54:777–789, 2003.

[66] A. Lodi, S. Martello, and D. Vigo. Models and bounds for two-dimensional level packing problems. *Journal of Combinatorial Optimization*, 8:363–379, 2004.

[67] S. Martello, M. Monaci, and D. Vigo. An exact approach to the strip-packing problem. *INFORMS Journal on Computing*, 15(3):310–319, 2003.

[68] S. Martello, D. Pisinger, and D. Vigo. The three-dimensional bin packing problem. *Oper. Res.*, 48(2):256–267, 2000.

[69] T. C. Martins. *Two-dimensional packing utilising evolutionary algorithms and other meta-heuristic methods*. PhD thesis, University of Wales, Cardiff, UK, 2000.

[70] T. C. Martins and M. S. G. Tsuzuki. Simulated annealing applied to the irregular rotational placement of shapes over containers with fixed dimensions. *Expert Systems with Applications*, 37:1955–1972, 2010.

[71] F. K. Miyazawa and Y. Wakabayashi. Three-dimensional packings with rotations. *Computers and Operations Research*, 36:2801–2815, 2009.

[72] M. Mongeau and C. Bes. Optimization of aircraft container loading. *IEEE Transactions on Aerospace and Electronic Systems*, 39:140–150, 2003.

[73] R. Morábito, M. Arenales, and V. F. Arcaro. An and-or-graph approach for two-dimensional cutting problems. *European J. Operational Research*, 58:263–271, 1992.

[74] R. Morábito and S. Morales. A simple and effective recursive procedure for the manufacturer's pallet loading problem. *Journal of the Operational Research Society*, 49:819–828, 1998.

[75] N. Ntene and J.H. van Vuuren. A survey and comparison of guillotine heuristics for the 2d oriented offline strip packing problem. *Discrete Optimization*, 6:174–188, 2009.

[76] J. F. Oliveira, A. M. Gomes, and J. S. Ferreira. Topos – a new constructive algorithm for nesting problems. *OR Spectrum*, 22:263–284, 2000.

[77] F. G. Ortmann, N. Ntene, and J. H. van Vuuren. New and improved level heuristics for the rectangular strip packing and variable-sized bin packing problems. *European Journal of Operational Research*, 203:306–315, 2010.

[78] J. Puchinger and G. R. Raidl. Models and algorithms for three-stage two-dimensional bin packing. *European J. Operational Research*, 183(3):1304–1327, 2007.

[79] T. A. Queiroz, F.K. Miyazawa, and Y. Wakabayashi. Heurísticas para o problema da mochila 2d não-guilhotinada ilimitada. In *Anais do CLEI 2010 - XXXVI Conferência Latino-americana de Informática*, page 14pgs, Assunção, Paraguai, 2010.

[80] T. A. Queiroz, F.K. Miyazawa, Y. Wakabayashi, and E.C. Xavier. Algoritmos para os problemas da mochila e do corte de estoque tridimensional guilhotinado. In *Anais do XLI Simpósio Brasileiro de Pesquisa Operacional*, page 12pgs, Porto Seguro, BA, 2009.

[81] M. C. Riff, X. Bonnaire, and B. Neveu. A revision of recent approaches for two-dimensional strip-packing problems. *Engineering Applications of Artificial Intelligence*, 22:823–827, 2009.

[82] W. F. Riley, L. D. Sturges, and D. H. Morris. *Statics and Mechanics of Materials: An Integrated Approach*. John Wiley & Sons, 2 edition, 2001.

[83] G. Scheithauer. Equivalence and dominance for problems of optimal packing of rectangles. *Ricerca Operativa*, 27:3–34, 1997.

[84] F. L. S. Silva, T. A. Queiroz, and F.K. Miyazawa. Um algoritmo branch-and-cut para o problema de empacotamento em faixa bidimensional sujeito a restrição de ordem e a estabilidade dos objetos. In *Anais do CLEI 2010 - XXXVI Conferência Latino-americana de Informática*, page 14pgs, Assunção, Paraguai, 2010.

[85] J. L. Castro Silva, N. Y. Soma, and N. Maculan. A greedy search for the three-dimensional bin packing problem: the packing static stability case. *International Transactions in Operational Research*, 10(2):141–153, 2003.

[86] G. Wäscher, H. Haussner, and H. Schumann. An improved typology of cutting and packing problems. *European Journal of Operational Research*, 183(3):1109–1130, 2007.

[87] D. F. Zhang, S. D. Chen, and Y. J. Liu. An improved heuristic recursive strategy based on genetic algorithm for the strip rectangular packing problem. *Acta Automatica Sinica*, 33:911–916, 2007.